コンピュータサイエンス・シリーズ

分散オペレーティングシステム

(東京大学)理学博士 プラディープ・K・シンハ 著
翻訳主幹 静岡大学教授・工学博士 水野　忠則
工学博士 井手口哲夫
工学博士 福岡　久雄 訳
工学博士 宮西洋太郎
工学博士 佐藤　文明

科学技術出版

コンピュータサイエンス・シリーズ

分散オペレーティングシステム

(東京大学)理学博士 プラディープ・K・シンハ 著

翻訳主幹 静岡大学教授・工学博士 水野 忠則

工学博士 井手口哲夫

工学博士 福岡 久雄 訳

工学博士 宮西洋太郎

工学博士 佐藤 文明

科学技術出版

Distributed Operating Systems
Concept and Design

by Pradeep K. Sinha
Centre for Develpment of Advanced Computing

ISBN4-87653-008-4

SciTech Press, Inc.
Hongo 2-chome 16-12, Bunkyoku,
Tokyo 113-0033
info@kagaku.co.jp
http://www.kagaku.co.jp/
FAX: (+81) 3-3818-7888

Printed in Japan

Printed in Japan

はじめに

動機

既存のメインフレームに比べ、マイクロプロセッサに基づいたワークステーションは驚く程、価格性能費が向上し、またネットワーク技術の発展は、大変魅力ある分散コンピューティングシステム環境を構築してきた。このようなシステムを実現するハードウェア上の問題点に関しては、深く検討されてきたが、多くの問題点があり、分散オペレーティングシステムを広く利用できることが期待されてきた。幸いなことに、研究機関や産業界における近年の研究開発によって、分散オペレーティングシステムを開発するための概念や設計事項が有効になってきた。さらに、分散オペレーティングシステムは、研究機関だけでなく、商業的にも利用可能となってきた。

分散コンピューティングシステムが増えてきたことによって、分散オペレーティングシステムについて学ぶことは、コンピュータサイエンス及びコンピュータ工学の学生にとって、ますます重要になってきた。その結果、多くの大学が大学院レベルにおいて、分散オペレーティングシステムに関して授業を行うようになってきており、さらに、学部学生のオペレーティングシステムのコースにおいても、分散オペレーティングシステムにおける基本的な概念と設計原理が組み込まれてきている。

しかしながら、分散オペレーティングシステムに対し、内容をわかりやすく、しっかりと解説しているよい教科書が欠落していた。最近出版されたいくつかの本を除いて、この分野のほとんどの本は、一つの研究分野に絞って解説している。このため、教官と学生にとって、分散オペレーティングシステムの全体的なイメージ作りは、現時点では複雑で、時間を要するものである。さらに、各種の調査研究や設計事項を見極めるために、分散オペレーティングシステムの全体的な構造を知りたかったり、研究をはじめようとする者、さらにコン

ピュータの専門家は、これらの目的に対し、適切な教科書を見つけることは困難であった。

これらの理由によって、私は、分散オペレーティングシステムに関する教科書の準備に向かって、研究を開始することを決めた。私の主要な目的は、読者がこの分野について、より高度な理解を得るために十分な背景を得ることができるように、分散オペレーティングシステムの最新の技術をわかりやすく、明快に解説することである。

概要

本書は、分散オペレーティングシステムに横たわる基本的な概念と設計原理を分かり易く説明しており、特定の分散オペレーティングシステムやハードウェアに限定していない。したがって、各種の分散オペレーティングシステムに適用可能な、基本的な概念と設計原理が、一般的に記述されている。なお、事例研究に関しては、実際の分散オペレーティングシステムについても触れている。

本書の題材は、主として研究調査文献を参照して取り入れられている。この分野で利用可能なほとんどの研究文献の中から、理論的なものより、実際のシステムの実践的なものの概念に特に重点をおいて、選択している。

各章においては、テキストの中の題材についての理解度をチェックしたり、研究を刺激することを意図した適切な演習問題を掲載している。

通常の参考文献に加え、インターネットのオンライン参考文献への選択ポインタのリストが、各章の話題についてより詳細な題材を読者に紹介している。

本書では、自然の形で勉強しやすく、解説風に表している。

内容

第1章では、分散コンピューティングシステム、分散オペレーティングシステム、及び分散オペレーティングシステムを特徴付ける点について、まず解説している。分散コンピューティング環境 (Distributed Computing Environment: DCE) の簡単な解説も行っている。DCE は、本書の幾つかの章において、キーとなる事例研究で紹介されている。

第2章では、コンピュータネットワークの簡単な紹介を行うとともに、最新のネットワーク技術を述べる。

第3、第4、第5章では、分散コンピューティングシステムにおける情報交

換において使用される各種の通信技術について述べる。特に、これら3つの章は、プロセス間通信機構の設計に必要となる技術や、この技術にとって共通的に必要となる具体的な項目について述べている。第3章は、メッセージパッシング機構を取り扱っている。第4章は、遠隔手続き呼び出し機構を取り扱っている。第5章は、プロセス間通信における分散共有メモリ機構を取り扱っている。第6章では、時間同期、相互排他、デッドロック、選任アルゴリズムといった分散システムにおいて重要な同期について述べている。

第7章では、分散システムにおける資源管理において共通的に使用される方式について述べている。

第8章では、プロセス管理方式について述べている。特に、プロセス移行機構とスレッド支援機構を述べている。

第9章では、分散システムに対するファイルシステムに関して述べている。

第10章では、分散システムにおけるオブジェクトの名前付けと位置に関して方式と機構について述べている。

第11章では、分散システムにおけるセキュリティ方式とセキュリティ機構について述べている。

最後に、第12章では、具体的な分散オペレーティングシステムでもって、これまでの章で述べた概念に関連する4つの存在する分散オペレーティングシステムの事例研究について述べている。

対象読者

本書は、分散オペレーティングシステムの簡明で、分かり易い解説を望んでいる人を対象としている。

分散オペレーティングシステムの授業における理想的な教科書として、使用可能である。また、集中オペレーティングシステムの基本的な概念と設計方式に加え、分散オペレーティングシステムについてもカバーする必要がある大学学部高学年及び大学院におけるオペレーティングシステムの教科書としても使用できる。

本書は、分散オペレーティングシステムについて学ぶ必要があったり、分散オペレーティングシステム又は分散アプリケーションシステムの設計と開発を行おうとしているシステム管理者、専門ソフトウェア技術者、コンピュータ科学者／研究者によって、自習用テキストとしても使用できる。

高度な研究者には、特定の話題をさらに深めるために、各章の終わりに掲載

している豊富な参考文献のリストと、インターネットのオンライン参考文献ポインタも有効であろう。

幅広い読者によって単純かつ容易に理解できるように十分に考慮はしているものの、読者は、初歩的なコンピュータアーキテクチャの知識を有し、標準的なオペレーティングシステムのテキストで基本的な集中型オペレーティングシステムを学んでいることは仮定している。

インターネット上の参考文献ポインタについて

章末の参考文献リストに示す数多くの参考文献に加え、インターネットを利用したオンライン参考文献ポインタを本書では掲載している。これらのポインタの目的は、次の二つである。

1. 章末の参考文献リストは、選択された参考文献だけを載せてある。その章にカバーされている話題に関する数多くの参考文献は、スペースの関係上、各章の参考文献リストには含まれていない。ポインタは、このような参考文献がどこにあるかを知りたい読者によって利用されてよい。

2. 章末の参考文献リストは、既に発刊されて文献だけを示している。分散オペレーティングシステムは、現在研究が積極的に行われており、大変多くの新しい文献が毎月発刊されている。インターネット上のオンライン参考文献は、時々刻々改訂されており、興味ある読者は、本書が発刊された後に、発刊された文献の位置を示すポインタを使用してよい。このように、本書に含まれている情報に加え、本書はさらに本書がカバーしている話題とその関連話題に関する今進みつつある活動状況を知るための道を読者に提供もする。

オンライン参考文献へのポインタの章末リストは、完全に網羅されてはいないことに注意されたい。私が知っていて、興味の参考文献の位置を容易に使用できうると思われるオンライン参考文献に対してのみポインタを提供した。さらに、オンライン参考文献に対する多くのミラー (同じ参考文献が複数のサイトに存在している) が時にある。このようなケースでは、参考文献に対する一つのポインタのみ提供した。ほとんどのオンライン参考文献は、オンライン文献ではなく、文献へのオンライン参照であることも注意されたい。オンライン参考文献で参照されるかなりの文献は、ハードコピーだけである。しかしながら、幾つかの参照される文献はインターネット上で、オンライン版が存在して

いる。このような文献に対しては参考文献は、文献のオンライン版へのポインタを示す URL(Uniform Resource Locators) を通常、含んでいる。もし、読者が URL を含む参考文献を見つけたならば、対応する文献のオンライン版への URL アクセスが続く。

謝辞

直接、間接を問わず、多くの人が本書の刊行に携わった。まず、本書は、分散コンピューティングシステム及び分散オペレーティングシステムの分野に貢献してきた研究者の成果に基づいており、それらの方々に感謝したい。

私の博士課程の指導教官である前川守先生に、分散オペレーティングシステムの分野での研究の機会を与えていただいたおかげで、本書が完成できた。

Galaxy 分散オペレーティングシステムプロジェクトのメンバーである Kentaro Shimizu、Xiaohua Jia、Hyo Ashishara、Naoki Utsunomiya、Hirohiko Nakano、Kyu Sung Park、Jun Hanano の皆さんには、活発な議論を経て、私の分散オペレーティングシステムに関する知識向上に役立った。

参考文献を収集し、そしてそれらをインターネット上で利用可能にしたすべての人々の尽力無しには、章末のインターネット参考文献へのポインタを掲載することはできなかった。

私の草稿原稿に幾人の匿名の査読者によって、章構成、カバーする範囲、タイプミス、明確でないテキスト部分等に関して、貴重なフィードバックをしていただいた。IEEE Press における編集担当である Denise Phillip は、本書のためにいろいろな方法で、すばらしい仕事をしていただいた。

IEEE 出版ディレクタ Dudley Kay、上級アクイジッションエディタ John Griffin、査読コーディネータ Lisa Mizrahi、Lisa Dayne は、本書を全体的により良くした点と、時宜にあって刊行に対して、多大にお世話になった。

最後に、完全な手書き原稿を電子的に入力した妻 Priti に感謝する。また、彼女にはこの長いプロジェクトの全体の期間中、継続的な忍耐と犠牲を強い、これに感謝する。彼女の親愛なる支援と理解なくして、このプロジェクトを完了の成功はあり得なかったであろう。

Pradeep K. Sinha

訳者まえがき

コンピュータはより安く、小さく、高性能となり、またコミュニケーションはより早く、高品質で、安く利用できるようになり、コンピュータとコミュニケーションの統合化が現実なものとなってきた。

当初コンピュータとコンピュータは単につながるだけで、かなりの革新であったが、時が経つにつれ、コンピュータとコンピュータが有機的につながり、全体が一元的なシステムとなることが要求されるようになってきた。

オペレーティングシステムは、ハードウェアとアプリケーションを仲介し、統合化を可能とするものであり、近年ソフトウェアが高度化するにつれ、ますます重要になってきた。オペレーティングシステムには、単体で動作するスタンドアローン型のオペレーティングシステム、それにネットワーク機能を付加したネットワークオペレーティングシステム、CPUを複数使用したマルチプロセッサオペレーティングシステムなどが存在するが、本書で紹介する分散オペレーティングシステムは、その中で最も進化したものであり、ネットワーク機能と純粋のオペレーティングシステム機能が完全に一体化したものである。

そのため、分散オペレーティングシステムは、各種の概念が組み合わされ、かつその概念は新しく、一般的でないものが多くあり、その全体像を把握することはかなり難解であった。

本書は、分散オペレーティングシステムのすべてを網羅し、かつ体系的にとらえ、分散オペレーティングシステムが何であるかを明確に解説している。そのため、参考文献は充実しており、特にインターネットのホームページ参照ポインタによる参考文献が記載されている。これにより、現時点での分散オペレーティングシステム技術が容易に入手できるようになっており、さらに、インターネットを利用することによって、本書以降の最新の状況をも入手することが可能となっている。さらに、本書の最後には豊富な演習問題が載っており、それを解くことにより、レベルの高い理解度が得られるものと思われる。

本書を勉強することによって、ネットワークを介した高度なコンピュータシステムを実現する各種の技術を取得でき、実際の実装、そして新たな技術への展開をはかることが可能となる。

なお、本書の訳は、井手口哲夫博士第1章～第4章、佐藤文明博士が第5章と第9章、宮西洋太郎博士が第6章～第8章と第10章、福岡久雄博士が第11章と第12章を分担し、全体を私が監修した。また、本書訳の素案作成には、三菱電機株式会社情報技術総合研究所大谷治之氏に協力いただいた。また、静岡大学情報学部石原進博士、山田耕史技官、王新さん他水野研究室の学生及び福田孝子さんには編集上多大の尽力をいただいた。関連の皆様方に感謝の意を表する。

1999年　工学博士　水野忠則

省略語

AAL ATM Adaptation Layer (ATM アダプテーション層)

ACL Access Control List (アクセス制御リスト)

AFS Andrew File System (アンドリューファイルシステム)

ANSI American National Standards Institute (米国標準化研究所)

API Application Programming Interface (アプリケーションプログラムインタフェース)

APPN Advanced Peer-to-Peer Networking (高度ピアツーピアネットワーキング)

ARP Address Resolution Protocol (アドレスリゾルーションプロトコル)

ARPANET Advanced Research Projects Agency NETwork (高度研究プロジェクトエージェンシーネットワーク)

ASN.1 Abstract Syntax Notation (抽象構文記法)

ATM Asynchronous Transfer Mode (非同期転送モード)

BIOS Basic Input Output System (基本入出力システム)

B-ISDN Broadband Integrated Services Digital Network (広帯域統合デジタルサービスネットワーク)

CBR Constant Bit Rate (固定ビットレート)

CCITT International Telegraph and Telephone Consultative Committee (国際電信電話通信機構)

CDS Cell Directory Service/Server (セルディレクトリサービス／サーバ)

CERN European Centre for Nuclear Research (欧州核研究センター)

CFS Cedar File System (シーダファイルシステム)

CICS Customer Information Control System (顧客情報管理システム)

CLP Cell Loss Priority (セル損失優先度)

CMH Chandy-Misra-Hass

CMIP Common Management Information Protocol (共通管理情報プロトコル)

COOL Chorus Object-Oriented Layer (コーラスオブジェクト指向層)

CSMA/CD Carrier Sense Multiple Access with Collision Detection (衝突検出機構付き搬送波センス多重アクセス)

CSRG Computer Systems Research Group (コンピュータシステム研究グループ)

DCE Distributed Computing Environment (分散コンピューティング環境)

DDLCN Distributed Double-Loop Computer Network (分散 2 重化ループコンピュータネットワーク)

DEC Digital Equipment Corporation (デジタル社)

DES Data Encryption Standard (データ暗号化標準)

DFS Distributed File Service (分散ファイルサービス)

DI Directory Identifier (ディレクトリ識別子)

DIB Directory Information Base (ディレクトリ情報ベース)

DIT Directory Information Tree (ディレクトリ情報木)

DME Distributed Management Environment (分散管理環境)

DN Distinguished Name (識別名称)

DNS Domain Name/Naming Service/System (ドメイン　名前／名前付けサービス／サーバ)

DoD Department of Defense (米国国防省)

DSM Distributed Shared Memory (分散共有メモリ)

DSVM Distributed Shared Virtual Memory (分散共有仮想メモリ)

DTS Distributed Time Service (分散時間サービス) item[Email] Electronic Mail (電子メール)

ERB Expanding Ring Broadcast (拡張リングブロードキャスト)

FDDI Fiber Distributed Data Interface (ファイバ分散データインタフェース)

FIFO First-In First-Out (先入れ先出し)

FLIP Fast Local Internet Protocol (高速ローカルインターネットプロトコル)

FTP File Transfer Protocol (ファイル転送プロトコル)

Gbps Gigabits per second (ギガビット／秒)

GDA Global Directory Agent (グローバルディレクトリエージェント)

GDS Global Directory Service/Server (グローバルディレクトリ　サービス／サーバ)

GEOS Geostationary Operational Environmental Satellites (環境静止衛星)

GFC Generic Flow Control (汎用フロー制御)

GNS Global Name Service (グローバル名前サービス)

GNU Gnu's Not Unix

GPS Global Positioning System (グローバル位置システム)

HEC Header Error Control (ヘッダ誤り制御)

HRPC Heterogeneous Remote Procedure Call (異種遠隔手続き呼び出し)

IBM International Business Machines (IBM 社)

ICMP Internet Control Message Protocol (インターネット制御メッセージプロトコル)

IDL Interface Definition Language (インタフェース定義プロトコル)

IEEE Institute of Electrical and Electronics Engineers (米国電気電子技術協会)

IETF Internet Engineering Task Force (インターネットエンジニアリングタスクフォース)

IMS Information Management Systems (情報管理システム)

INRIA Institute National de Recherche en Informatique et Automatique (国立情報自動化研究所)

IP Internet Protocol (インターネットプロトコル)

IPC Inter-Process Communication (プロセス間通信)

ISDN Integrated Services Digital Network (デジタルサービス統合網)

ISO International Standards Organization (国際標準化機構)

ITU International Telecommunications Union (国際電気通信連合)

KDBMS Kerberos Database Management Server (ケルベロスデータベース管理サービス)

KDC Key Distribution Center (鍵配布センタ)

LAN Local Area Network (ローカルエリアネットワーク)

LEC LAN Emulation Client (LAN エミュレーションクライアント)

LES LAN Emulation Server (LAN エミュレーションサーバ)

LRPC Lightweight Remote Procedure Call (ライトウエイト遠隔手続き呼び出し)

MAN Metropolitan Area Network (メトロポリタンエリアネットワーク)

MBone Multicast Backbone (マルチキャストバックボーン)

Mbps Megabits per second (メガビット／秒)

MIG Mach Interface Generator (マークインタフェースジェネレータ)

MMU Memory Management Unit (メモリ管理単位)

MTBF Mean Time Between Failures (故障平均間隔)

MTU Maximum Transfer Unit (最大転送単位)

NCA Network Computing Architecture (ネットワークコンピューティングアーキテクチャ)

NCS Network Computing System (ネットワークコンピューティングシステム)

NFS Network File System (ネットワークファイルシステム)

NIC Network Information Center (ネットワーク情報センター)

NIST National Institute for Standards and Technology (国立標準技術研究所)

NRMB Non-Replicated Migrating Block (非複製移動ブロック)

NRNMB Non-Replicated Non-Migrating Block (非複製非移動ブロック)

NSAP Network Service Access Point (ネットワークアクセスサービス点)

NTP Network Time Protocol (ネットワーク時間プロトコル)

NUMA Non-Uniform Memory Access (非均一メモリアクセス)

OC-n Optical Carrier level n (光搬送波レベル n)

OLTP On Line Transaction Processing (オンライントランザクション処理)

OPC Output Port Controller (出力ポート制御装置)

OSF Open Software Foundation (オープンソフトウェアファンデーション)

OSI Open Systems Interconnection (開放型システム間相互接続)

PCB Process Control Block (プロセス制御ブロック)

PEM Privacy Enhanced Mail (プライバシ拡張メール)

PKM Public Key Manager (公開鍵管理者)

PMD Physical Medium Dependent (物理媒体依存)

POSIX Portable Operating System Interface for Computer Environments (コンピュータ環境用ポータブルオペレーティングシステムインタフェース)

PRAM Pipelined Random Access Memory (パイプラインランダムアクセスメモリ)

PSE Packet Switching Exchange (パケット交換機)

PTI Payload Type Identifier (ペイロード型識別子)

RARP Reverse Address Resolution Protocol (逆アドレスリゾルーションプロトコル)

RDN Relative Distinguished Name (相対識別名称)

RFS Remote File Server (遠隔ファイルサーバ)

RFT Request For Technology (技術養成)

RMB Replicated Migrating Block (複製移動ブロック)

RNMB Replicated Non-migrating Block (複製非移動ブロック)

RPC Remote Procedure Call (遠隔手続き呼び出し)

RPCL Remote Procedure Call Language (遠隔手続き呼び出し言語)

RR Round-Robin (ラウンドロビン)

RSA Rivest-Shamir-Adleman

RSS Research Storage System (リサーチ格納システム)

SDH Synchronous Digital Hierarchy (同期デジタルハイアラキ)

SEAL Simple and Efficient Adaptation Layer (簡易／効果アダプテーション層)

SLIP Serial Line Internet Protocol (シリアル回線インターネットプロトコル)

SMTP Simple Mail Transfer Protocol (簡易メール転送プロトコル)

SNA System's Network Architecture (システムネットワーク体系)

SNMP Simple Network Management Protocol (簡易ネットワーク管理プロトコル)

SONET Synchronous Optical NETwork (同期光ネットワーク)

STS-n Synchronous Transport Signal level n (同期転送信号レベル n)

TC Transmission Convergence (転送コンバージェンス制御)

TCF Transparent Computing Facility (透過コンピューティング機能)

TCP Transmission Control Protocol (転送制御プロトコル)

TFTP Trivial File Transfer Protocol (簡易ファイル転送プロトコル)

TI-RPC Transport Independent-Remote Procedure Call (トランスポート非依存ー遠隔手続き呼び出し)

TP Transport Protocol (トランスポートプロトコル)

TWFG Transaction Wait-For-Graph (トランザクション待ち状態グラフ)

UCP Unilateral Commit Protocol (一方コミットプロトコル)

UDP User Datagram Protocol (ユーザデータグラムプロトコル)

UDS Universal Directory Service (ユニバーサルディレクトリサービス)

UNI User Network Interface (ユーザネットワークインタフェース)

UTC Coordinated Universal Time (協定世界時)

UUID Universally Unique Identifier (ユニバーサリ固有識別子)

VBR Variable Bit Rate (可変ビットレート)

VCI Virtual Channel Identifier (仮想チャネル識別子)

VM Virtual Memory (仮想メモリ)

VMTP Versatile Message Transfer Protocol (高機能メッセージ転送プロトコル)

VPI Virtual Path Identifier (仮想パス識別子)

WAIS Wide Area Information Servers (広域情報サーバ)

WAN Wide Area Network (広域ネットワーク)

WFG Wait For Graph (待ち状態グラフ)

WWW World-Wide Web (ワールドワイドウエブ)

X-IPC eXtented Inter-Process Communication (拡張プロセス間通信)

XDR eXternal Data Representation (拡張データ表現)

XDS X/Open Directory Server (X オープンディレクトリサーバ)

XNS Xerox Networking System (ゼロックスネットワーキングサービス)

XOM X/Open Object Management (X オープンオブジェクト管理)

目　　次

第1章

基本技術

1.1 分散コンピューティングシステムとは

過去20年間において、マイクロエレクトロニクス技術の発展は、高速でかつ低価格なプロセッサをもたらし、また通信技術の発展により経済性のある高能率なコンピュータネットワークが出現してきた。これらの技術により価格性能比は向上し、従来の単一の高速プロセッセ構成に代わって複数プロセッサの相互接続構成を可能にした。

複数プロセッサの相互接続によるコンピュータ構成法には、基本的に次の2形態がある。

1. 密結合 (**Tightly coupled systems**) システム: このシステム形態では、全てのプロセッサが共有する1つの主メモリ (アドレス空間) が存在する (図1.1(a) 参照)。例えば、あるプロセッサが値100の情報をメモリxに書くと、他のプロセッサはメモリxから値100を読み出すことができる。すなわち、このシステムにおいてはプロセッサ間の通信は共有メモリを介して行われる。

2. 疎結合 (**Loosely coupled systems**) システム: このシステム形態では、全てのプロセッサに対する共有メモリはなく、各プロセッサでローカルメモリを持つ (図1.1(b) 参照)。例えば、あるプロセッサが値100の情報をメモリxに書くと、この書き込み動作ではそのプロセッサのローカルメモリの内容を変えるだけで、他のプロセッサには何ら影響を与えない。したがって、他のプロセッサがメモリxの内容を読み出しても以前そのプロセッサが書いた値が読み出される。このシステムにおいて全てのプロセッサ間通信は、プロセッサを相

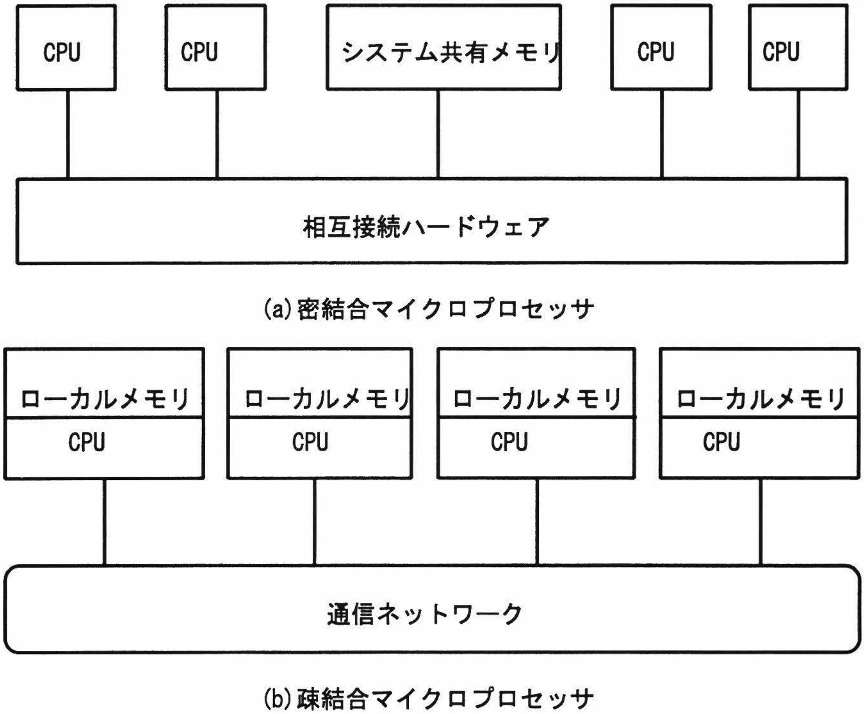


図 **1.1**　密結合と疎結合マイクロプロセッサシステムの相違

互に接続しているネットワークを介するメッセージの交信で行われる。

一般に、密結合システムは**並列処理システム (parallel processing systems)**、疎結合システムは**分散コンピューティングシステム (distributed computing systems)** 又は単に分散システムとも呼ばれる。本書では、「分散システム」という用語は分散オペレーティングシステム (1.5 節参照) を利用する分散コンピューティングシステムに対してのみ使用する。したがって、1.5 節において「分散システム」を定義するまでは、「分散コンピューティングシステム」は疎結合システムを表す。密結合システムと比べて、分散コンピューティングシステムのプロセッサは地理的に遠く離れた地域にまたがって相互に接続することができる。さらに、密結合システムでは構成できるプロセッサの数は共有メモリ容量の制限から通常小さくなるが、分散コンピューティングシステムではメ

モリ空間は自由に拡張可能であり、プロセッサの数はほとんど制限されない。すなわち、分散コンピューティングシステムは基本的にローカルメモリと周辺機器を持つプロセッサが通信ネットワークを介して相互に接続したシステムであり、このシステムの任意の2つのプロセッサ間通信は通信ネットワーク上でのメッセージパッシングによって行われる。あるプロセッサに注目すると、そのプロセッサの資源は**ローカル (local)** にあり、その他のプロセッサの資源は**遠隔 (remote)** にある。また、プロセッサ及びその資源は一般に分散コンピューティングシステムの**ノード (node)** 又は**サイト (site)**、あるいは**マシン (machine)** と呼ばれる。

1.2 分散コンピューティングシステムの発展

初期のコンピュータは、大変高価で (数百万ドル)、大変大掛かりなものであった (大きな部屋が必要であった)。コンピュータは貴重なものであり、大学や企業の研究所でのみ利用されていた。この時代のコンピュータは、オペレータがコンソールから実行を起動し、一般のユーザは操作できなかった。プログラマは、プログラミングをすると、例えばパンチカードでコンピュータセンタに処理を依頼していた。ジョブを実行するために、まずオペレータが必要な環境 (テープの設定、カード読み取り装置からパンチカードの読み込みなど) を設定した。このようにして実行されると、結果が印刷され、プログラマに渡された。

ジョブを設定するための時間は初期のコンピュータでは実際大変な問題であり、貴重な中央処理装置 (CPU) 時間の多くを無駄にしていた。その後、幾つかの新しい方式がコンピュータの CPU 利用率を高めるために 1950 年代及び 1960 年代に生まれた。注目するものとして、処理の前に同じような要求を持つジョブを一括して処理する方法、ジョブを順番に自動的に処理する方法、バッファリングやスプーリングの方式に基づく**オフライン処理 (off-line processing)** や**マルチプログラミング (multiprogramming)** があった。同じようなジョブをまとめて処理する**バッチ処理 (batching)** は、新しいジョブを起動する直前に必ず実行環境の設定をオペレータが行う必要があるためほとんど CPU 利用率の改善には寄与しなかった。ジョブの始めと終わりを制御カードで定義する**自動ジョブ順次処理 (automatic job sequencing)** は、人の手間を省くことができたため CPU 利用率の改善をもたらした。オフライン処理は、独立した2つの装置で2つの動作を実行すること (I/O は一般に CPU よりかなり処理速度が遅い) によって CPU と入出力 (I/O) 動作を同時に行うことで CPU 利用率

を向上できた。最後に、マルチプログラミングは、CPU が常に実行すべき仕事があるようにジョブを構成することにより CPU 利用率を向上させることができた。

しかしながら、これらの方法は複数のユーザが直接コンピュータシステムを操作することや同時に資源を共有することはできなかった。したがって、次の動作が前の動作の結果に関係するような多くの短い動作から構成される相互作用を持つジョブは、冗長で、結果的に多くの時間を消費することになった。このような相互作用を持つジョブの例としてプログラムの開発やデバッグ作業がある。この問題を解決するための**タイムシェアリング (time sharing)** の方式は、1970 年代の始めになって導入された。初期のタイムシェアリングシステムは、複数のダム端末がメインコンピュータに接続される構成であった。これらの端末はメインコンピュータの部屋とは別の部屋に置かれた。これらの端末から複数のユーザは同時に相互作用を持つジョブを実行し、コンピュータシステムの資源を共有することができた。タイムシェアリングシステムでは、それぞれのユーザのジョブの一部分を短時間に実行し切り替えられるため、各ユーザはコンピュータを占有しているような印象を与えられた。このタイムシェアリング方式は 1960 年代の始めに紹介されたが、実現することが容易でなくかつ高価につくため 1970 年代の始めまでタイムシェアリングコンピュータシステムは一般に普及しなかった。

ハードウェア技術の発展によりコンピュータの小型化と処理の高速化が現実的なものとなり、従来の大規模なコンピュータの処理能力を越えてかつ小型で安価なコンピュータが現れた。このようなコンピュータは、**ミニコンピュータ (minicomputer)** と呼ばれた。

タイムシェアリングシステムの出現は、分散コンピューティングシステムへの最初のステップであり、分散コンピューティングシステムにおける 2 つの重要な考え方、すなわち複数のユーザによるコンピュータ資源の共有とメインコンピュータから離れた場所からのアクセスが示された。当初、タイムシェアリングシステムの端末はダム端末であったため、全ての処理はメインコンピュータシステムで行われていた。1970 年代のマイクロプロセッサ技術の発展によりダム端末はインテリジェント端末に取って代わり、オフライン処理とタイムシェアリングの方式が単一システム上に組み合わせて実現された。マイクロプロセッサ技術は急速に進展し、1980 年代初頭には**ワークステーション (workstation)** と呼ばれる個人用のコンピュータを産み出した。このワークステーションはミニコンピュータと同等の処理能力を持ち、かつ価格はミニコ

ンピュータよりかなり低額であった。例えば、Xerox PARC で開発された最初のワークステーション (Alto と呼ばれた) は、高解像度のモノクロ画面、マウス、主記憶 128 キロバイト、2.5 メガバイトのハードディスク、それに 2 マイクロ秒から 6 マイクロ秒でマシン命令を実行するマイクロプログラムの CPU から構成されていた。したがって、このワークステーションはタイムシェアリングシステムの端末として利用された。このようなタイムシェアリングシステムにおいては、ユーザジョブの処理の大半はワークステーションで行われ、メインコンピュータは多くのユーザのよって同時に利用することができた。メインコンピュータには共有される資源としてファイル、データベース及びソフトウェアライブラリが置かれた。

以上述べたように集中型のタイムシェアリングシステムでは、端末とメインコンピュータ間を通常のケーブルで接続するためにメインコンピュータ室から余り遠くに端末を置くことができなかった。一方、1960 年代末から 1970 年代始めおけるコンピュータネットワーク技術の発展により、2 つのキーとなるネットワーク技術である **LAN** (ローカルエリアネットワーク **(Local Area Network)**) と **WAN** (広域ネットワーク **(Wide Area Network)**) が出現した。LAN の技術は、約 10 メガビット／秒 (Mbps) のデータ伝送速度での通信を可能とし、LAN によって接続された複数のコンピュータをビルや学校のキャンパスに配置できるようになった。一方、WAN の技術は、約 56 キロビット／秒 (Kbps) のデータ伝送速度を可能とし、遠く離れたコンピュータ (異なる都市、国又は大陸) 間の通信を可能とした。最初の高速 LAN は 1973 年に Xerox PARC で開発された **Ethernet** であり、最初の WAN は 1969 年に米国国防省によって開発された **ARPAnet(Advanced Research Projects Agency Network)** である。その後、ネットワークの伝送速度は急速に向上し、1980 年代には 100 Mbps の LAN や 64 Kbps の WAN が可能となった。1990 年代に入った現在では、ネットワーク技術として新たな主要な技術である **ATM** (非同期転送モード **(asynchronous transfer mode)**) 技術が現れた。この ATM 技術は、まだ検討すべき課題が残されている技術である。しかし、この技術は広帯域ネットワークの実現を可能とするものであり、LAN および WAN において 1.2 ギガビット／秒 (Gbps) までのデータ伝送速度を提供できる。このような広帯域ネットワークは、将来の分散コンピューティングシステムが完全に新しいクラスの分散アプリケーション、すなわち音声、動画及び従来のデータを含む複合情報を処理できる**マルチメディアアプリケーション (multimedia application)** の実現を可能にする。このようなアプリケーションは、従来の

LAN や WAN 上では考えられないものであった。

コンピュータ技術とネットワーキング技術の融合によって 1970 年代の終わりに分散コンピューティングシステムが生まれた。このようなシステムを構築するためのハードウェアの課題は明確になっていたが、その時点での主な障害となっていたのは, 使い易く、かつ十分に能力を引き出すことができるソフトウェアが提供されなかったことであった。したがって、1970 年代の終わりから分散オペレーティングシステムの分野において大学や企業では注目する研究が行われた。これらの研究活動により、分散オペレーティングシステムの設計のための基本的な考え方が明確になってきた。この分野ではまだ多くの課題が残されており、活発に研究活動が行われているが, 商用の分散オペレーティングシステムが発表されはじめてきた。これらのシステムはそれまでに確立された基本的な技術に基づいている。本書では、これらの基本的な技術と分散オペレーティングシステムの設計と実現について述べる。幾つかの基本的な概念は、分散コンピューティングシステムのアプリケーション設計にも必要となるものであり, 本書は分散アプリケーションの設計者の利用にも適している。

1.3　分散コンピューティングシステムのモデル

分散コンピューティングシステムを構築するために各種のモデルが使用されている。これらのモデルは、大きくミニコンピュータ、ワークステーション、ワークステーション–サーバ、プロセッサ-プール及びハイブリッドの 5 つに分類される。これらについてその概要を次に述べる。

1.3.1　ミニコンピュータモデル

ミニコンピュータモデル (**minicomputer model**) は、集中型のタイムシェアリングシステムの単純な拡張である。図 1.2 に示すように、このモデルに基づく分散コンピューティングシステムは、通信ネットワークを介して接続された数台のミニコンピュータ (それらは大規模のスーパコンピュータでも良い) から構成される。各ミニコンピュータは、通常同時にログインする複数のユーザに対応できる。したがって、数台の端末が各ミニコンピュータに接続される。ユーザは、遠隔のミニコンピュータにアクセスすることによって特定のミニコンピュータにログインする。ネットワークによって現在ログインしているミニコンピュータ以外のミニコンピュータが持つ資源を使用することができる。

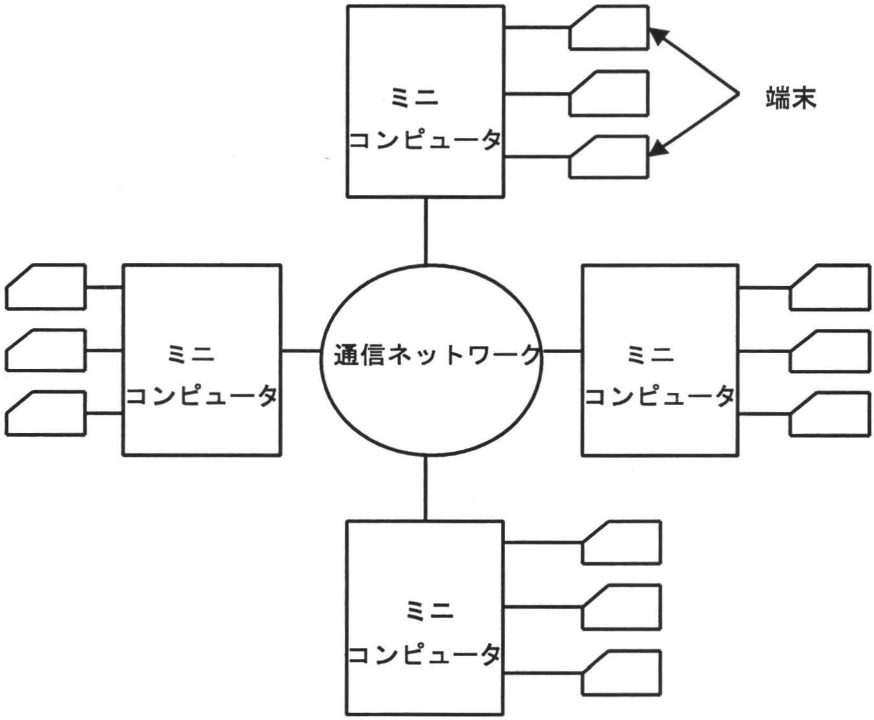


図 **1.2** ミニコンピュータモデルによる分散コンピューティングシステム

ミニコンピュータモデルは、共有資源(各種のタイプの情報データベースが共有される場合にそれぞれのタイプのデータベースは別々のマシンに置かれる)が遠隔のユーザから要求された時に有効である。

初期の **ARPAnet** は、ミニコンピュータモデルに基づく分散コンピューティングシステムの例である。

1.3.2 ワークステーションモデル

図 1.3 に示すように、ワークステーションモデル (**workstation model**) に基づく分散コンピューティングシステムは、通信ネットワークを介して接続された数台のワークステーションから構成される。企業のオフィスや大学の各学部では、個人用のコンピュータとしてディスクを持ったワークステーションがビルやキャンパスに数多く配置されている。このような環境では、多くの時間

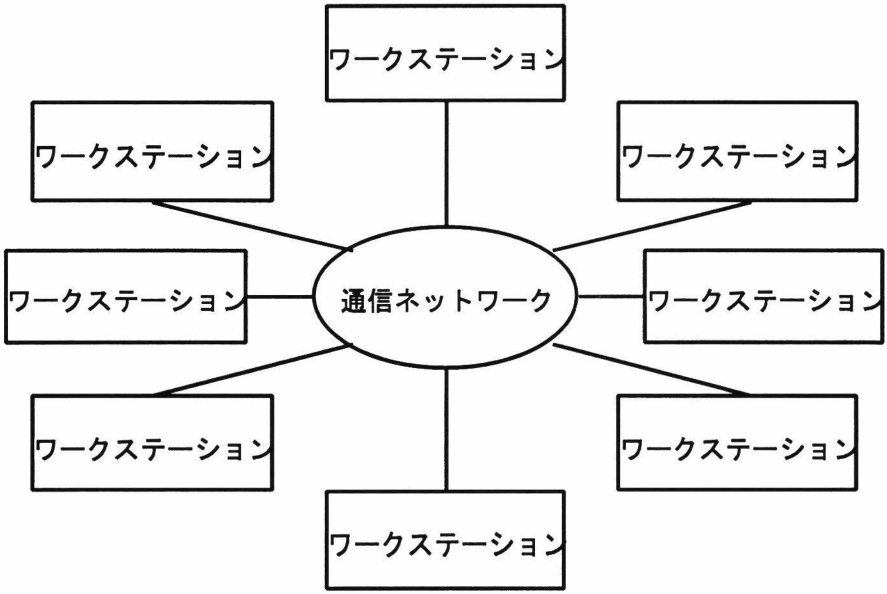


図 **1.3**　ワークステーションモデルによる分散コンピューティングシステム

(特に夜間) ワークステーションは空き状態であり、多くの CPU 時間が無駄になっている。したがって、このようなワークステーションモデルでは、自分のワークステーションが十分な処理能力を持たないユーザのために高速 LAN によってワークステーションを相互に接続し、他のワークステーションの処理能力を利用できるようにすることが望まれる。

このモデルでは、ユーザがホームと呼ばれるワークステーションにログインし、ジョブの実行を依頼すると、システムは依頼されたジョブをそのワークステーションで処理できないと判断すると、1 つ若しくは複数のプロセスを空いているワークステーションあるいは実行する能力があるワークステーションに依頼し、実行結果をそのユーザのワークステーションに戻す。

このモデルは一見可能に思われるが、幾つかの課題があり、実現することは簡単ではない。これらの課題として次のものがある [Tanenbaum 1995]。

1. 空いているワークステーションの検出方法
2. ワークステーション間でのプロセスの転送方法

3. 空いているワークステーションで他のワークステーションのプロセスを実行している時にユーザがログインした場合の遠隔プロセスの扱い

最初の2つの課題は、それぞれ第7章と第8章で述べる。3番目の課題に対する考え方を下記に3つ示す。

1. 最初の考え方は、ログインしたユーザのプロセスと一緒に遠隔プロセスがワークステーションの資源を共有する方法である。この方法の実現は容易であるが、個人用のコンピュータとしてのワークステーションの考え方に反する。なぜなら、遠隔プロセスとログインしたユーザのプロセスが同時に実行されるためログインしたユーザに対して応答性能が低下する。
2. 2番目の考え方は、遠隔プロセスを止める方法である。この方法の欠点は、これまでの遠隔の処理が失われ、ファイルシステムは一貫性を保持できなくなるため美しくない方法である。
3. 3番目の考え方は、遠隔プロセスをホームのワークステーションに戻し、実行を継続する方法である。この方法は、プロセスの途中での移行機構が必要となるため実現は難しい。この処理の途中での移行機構の定義及びその課題については第8章で述べる。

ワークステーションモデルの事例として、Spriteシステム [Ousterhout et al. 1988] と Xerox PARC で開発された実験システム [Shoch and Hupp 1982] がある。

1.3.3 ワークステーション-サーバモデル

ワークステーション-サーバモデル (workstation-server model) は、ディスクとローカルファイルシステムを持つ個人用のワークステーションのネットワークである。ローカルディスクを持つワークステーションを一般に**ディスクフルワークステーション (diskful workstation)** と呼び、ディスクがないものを**ディスクレスワークステーション (diskless workstation)** と呼ぶ。高速ネットワークの急速な普及により、ネットワーク環境ではディスクレスワークステーションが一般的になっており、分散コンピューティングシステムを構築する上ではワークステーションモデルよりもワークステーション-サーバモデルが導入されている。

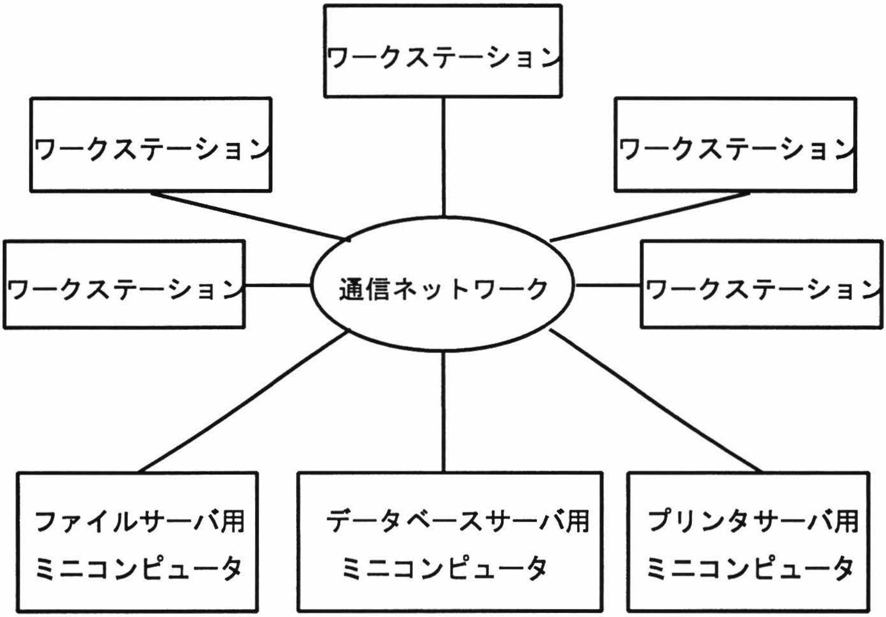


図 **1.4**　ワークステーションサーバモデルによる分散コンピューティングシステム

図 1.4 に示すように、ワークステーション-サーバモデルに基づく分散コンピューティングシステムは通信ネットワークを介して接続される少ないミニコンピュータと多くのワークステーション (一部にはディスクフルがあるが多くはディスクレス) から構成される。

ディスクレスワークステーションがネットワークに接続された場合、これらのワークステーションが使用するファイルシステムをディスクフルワークステーション若しくはディスクを持つミニコンピュータに実装することが必要となる。ミニコンピュータはファイルシステムのために使用され、1 台若しくは複数のミニコンピュータが接続される。他の目的のためにミニコンピュータが使用される場合があり、例えばデータベースサービスやプリントサービスがある。したがって、ミニコンピュータは各種のサービスを提供するためのサービスマシンとして使用される。このように、ワークステーション-サーバモデルではワークステーションに加えて、資源の管理や共有資源へのアクセスを実現するサーバプロセス (サーバ (**server**) と呼ばれる) を実行するための専用のマシン (専用のワークステーション) がある。

信頼性の向上や大規模化などのために、分散コンピューティングシステムにおいて特定の資源を管理するために複数のサーバが利用される。例えば、システムの全てのユーザのファイルを管理するために、複数のミニコンピュータがネットワークを介して相互に動作する複数のファイルサーバがある。このように、クライアントに提供されるサービスとサービスを提供するサーバとは別のものである。すなわち、サービス (**service**) は1つ若しくは複数のサーバによって提供されるものである。例えば、1つ若しくは複数のファイルサーバがユーザにファイルサービスを提供するために分散コンピューティングシステムにおいて利用される。

このモデルでは、ユーザは自分の**ホームワークステーション (home workstation)** にログインする。ユーザの処理を行う上で必要とされる通常の計算は、そのユーザのホームワークステーションで行われるが、特定のサーバ(例えば、ファイルサーバやデータベースサーバ) によって提供されるサービスがユーザの処理を行う上で必要となるとそのサーバへ要求が出され、その処理結果がユーザのワークステーションに戻される。したがって、このモデルでは、サーバマシンによって行われるサービスを得るためにそのサーバマシンにユーザの処理自体を移行する必要はない。

システム全体の性能の向上を目指して、ディスクフルワークステーションのローカルディスクが、一時的なファイルの格納、共有しないファイルの格納、まれに変更される共有のファイルの格納、仮想記憶におけるページ処理及び遠隔のデータのキャッシングのために使用される。

このワークステーションーサーバモデルは、ワークステーションモデルと比較して次のような利点がある。

1. 一般に、小容量な遅いディスクを持った数多くのディスクフルワークステーションよりも、大容量で高速なディスクを持つ数台のミニコンピュータをネットワークを介して利用する方が安価である。

2. ディスクレスワークステーションはシステム保守の観点からするとディスクフルワークステーションよりも扱い易い。ビルやキャンパスの各所に配置されている多くの小容量なディスクを保守するよりも数少ない大容量なディスクのバックアップやハードウェア保守を行う方が容易である。さらに、ソフトウェアを更新 (例えば、新機能ファイルサーバ) する場合でも、全てのワークステーションにソフトウェアを更新するよりも数台のファイルサーバにソフトウェアを更新する方が手間が省ける。

3. ワークステーション-サーバモデルでは、全てのファイルはファイルサーバによって管理されているため、ユーザはログインしているワークステーションに関わらず、他のワークステーションを使用し、かつファイルにアクセスすることができる融通性が与えられている。しかし、ワークステーションモデルでは、各ワークステーションがローカルファイルシステムを持っているためローカルのファイルアクセスと遠隔のファイルアクセスに対して異なった方法が必要となるため一貫性がなくなる。

4. ワークステーション-サーバモデルにおいては、サーバマシンのサービスを利用するために既に述べたような要求／応答プロトコルが主に使用される。そのため、ワークステーションモデルとは異なって、このモデルでは実現が難しいプロセスの移行機能が必要とされない。

要求／応答プロトコルは (request-response protocol) 、クライアントサーバモデル (client-server model) の通信として知られている。このモデルでは、クライアントプロセス (この場合ワークステーションに置かれる) がサーバプロセス (この場合ミニコンピュータに置かれる) にあるサービス、例えばファイルを読むために要求を送る。サーバは、要求を実行し、要求の処理結果をクライアントに返送する。

クライアントサーバモデルは、分散コンピューティングシステムにおいて情報や資源を共有するための効果的で汎用的なアプローチを提供する。これは、ワークステーション-サーバモデル形態のみならず各種のハードウェア及びソフトウェア環境における実現形態にも適用することができる。クライアントやサーバのプロセスが実行するコンピュータは、必ずしもワークステーションやミニコンピュータである必要はない。各種のものが利用でき、その違いは特になく、クライアントとサーバを同じコンピュータで実行することもできる。さらに、あるプロセスはクライアントとサーバの両者になることもできる。すなわち、サーバがクライアントとして他のサーバのサービスを利用することができる。

5. ワークステーションが遠隔プロセスを実行しないためサーバでの応答時間が保証できる。しかし、このモデルではワークステーションの空き時間を有効に利用するものではない。

ワークステーション-サーバモデルに基づく分散コンピューティングシステムの事例として V-System[Cheriton 1988] がある。

1.3.4 プロセッサプールモデル

プロセッサプールモデル (**processor-pool model**) は、大半の時間はほとんど処理能力を必要とせず、ある短時間に多くの処理の能力が必要となるような場合 (例えば、共通に使用する宣言文を変更した後に数多くのファイルから構成されるプログラムを再コンパイルする場合) を想定している。したがって、1つのプロセッサが1ユーザに割り当てられるワークステーションサーバモデルとは異なり、プロセッサプールモデルでは必要に応じてユーザが共有できるように複数のプロセッサがプールされている。プールされているプロセッサは、ネットワークに接続された数多くのミニコンピュータから構成される。各プロセッサは、それぞれメモリを持ち、分散コンピューティングシステムのシステムプログラムやアプリケーションプログラムをロードし実行することができる。

図1.5に示すようにプロセッサプールモデルでは、プールされているプロセッサは直接端末に接続されておらず、ユーザはネットワークに接続されている端末からシステムにアクセスする。これらの端末は、小型のディスクレスワークステーション若しくはX端末のようなグラフィック端末である。特殊なサーバ (実行サーバ (**run server**) と呼ばれる) はプール内のプロセッサを管理しており、ユーザの要求に従って割り当てる。ユーザが計算のためにジョブを投入すると、実行サーバによってそのジョブに適切な数のプロセッサを一時的に割り当てる。例えば、ユーザのジョブがそれぞれのセグメントとして独立に再配置可能なオブジェクトファイルにコンパイルできるようなn個のセグメントからなるプログラムであれば、プールからn個のプロセッサをn個のセグメントに割り当てることにより並列にコンパイルすることが可能となる。処理が完了すると、別のユーザが利用できるようにプロセッサはプールに戻される。

このプロセッサプールモデルでは、ホームマシンの概念はない。すなわち、ユーザは特定のマシンにログインするのではなく、システム全体にログインする。これは、他のモデルではユーザがホームマシン (例えば、ワークステーション又はミニコンピュータ) を持ち、それにログインし、基本的にそのユーザのプログラムをそのマシンで実行する方法と対照的である。

ワークステーションサーバモデルと比較して、プロセッサプールモデルは分散コンピューティングシステムにおいて処理能力の利用率は良い。これは、プロセッサプールモデルでは分散コンピューティングシステムの全ての処理能力がログインしているユーザに利用できるのに対して、ワークステーションサーバモデルでは幾つかのワークステーションが空いていても他のユーザのジョブ

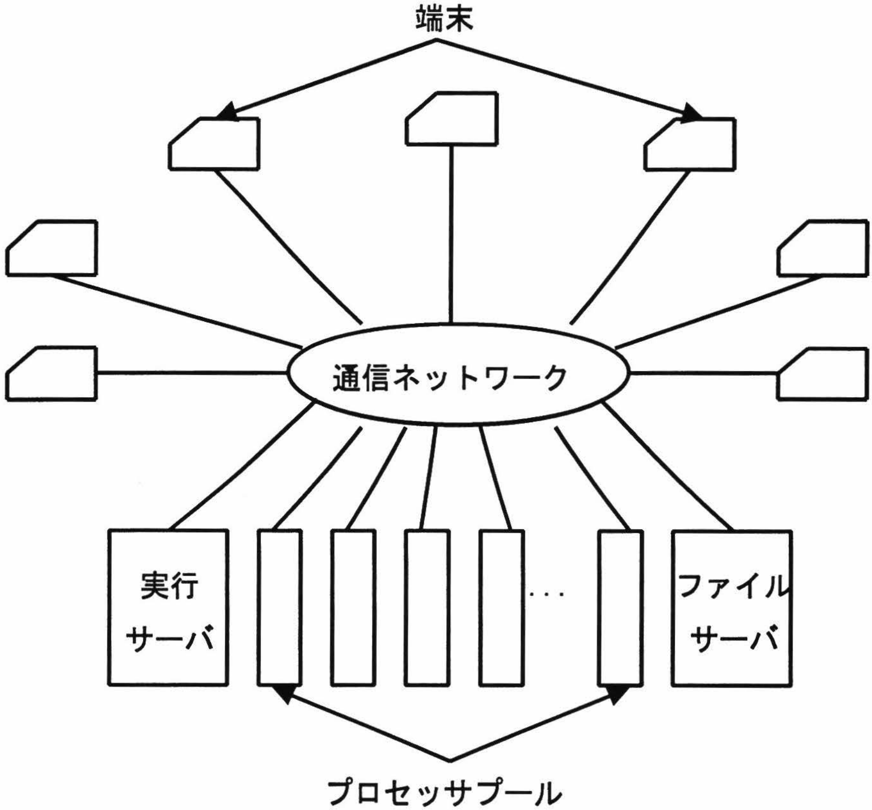


図 **1.5**　プロセッサプールモデルによる分散コンピューティングシステム

の処理に使えないためである。さらに、プロセッサプールモデルはワークステーションサーバモデルに比べてシステムのサービスを拡張するために新たなプロセッサの増設を必要としないなど柔軟性が高い。プールのプロセッサはユーザ数の増加に対して負荷が増加した場合や新しいサービスを追加する場合に、特定のサーバとして割り当てることができる。しかしながら、プロセッサプールモデルは高速に相互にやり取りを行うアプリケーション、特にグラフィックスやウィンドウシステムには不向きである。これは、ユーザのアプリケーションプログラムが実行するコンピュータと、システムと相互にやり取りを行うユーザの端末間の通信が低速であることに起因する。逆にワークステーションサーバモデルはこのようなアプリケーションに向いていると考えられる。

Amoeba [Mullender et al. 1990], Plan 9[Pike et al. 1990] 及びケンブリッジ分散コンピューティングシステム [Needham and Herbert 1982] は、プロセッサプールモデルに基づいた分散コンピューティングシステムの事例である。

1.3.5 ハイブリッドモデル

上述した4つのモデルの中で、ワークステーションサーバモデルが分散コンピューティングシステムを構築する上で最も幅広く利用されるモデルである。これは、大半のコンピュータユーザは単純な相互動作の処理、例えば編集処理、電子メールの送信や小さなプログラムの実行を行うためである。ワークステーションサーバモデルは、このような単純な利用形態では理想的なモデルである。しかし、処理の大きなジョブを実行するユーザグループがいる動作環境では、プロセッサプールモデルがより有効であり適している。

このようなワークステーションサーバモデルとプロセッサプールモデルの利点を兼ね備えるために**ハイブリッドモデル (hybrid model)** がして分散コンピューティングシステムを構築するために利用される。ハイブリッドモデルは、ワークステーションサーバモデルに基づいており、プロセッサプールが付加されたものである。プールのプロセッサは、ワークステーションには大きすぎる処理や効果的な実行を行うために複数のコンピュータを要求する処理のために動的に割り当てられる。処理そのものを効果的に実行することに加えて、相互作用が伴うような処理に対して応答性能を保証するためにこのハイブリッドモデルが使用される。しかし、ハイブリッドモデルはワークステーションサーバモデルやプロセッサプールモデルに比べてコスト高になる。

1.4 分散コンピューティングシステムの普及の理由

これまで述べた分散コンピューティングシステムのモデルから、分散コンピューティングシステムを構築することは従来の**集中型システム (centralized systems)** (1つの CPU、メモリ、周辺装置、1台若しくは複数台の端末から構成される) に比べてさらに複雑で困難であることがわかる。この複雑度が増している要因は、膨大な分散資源を効率良く利用するとともに管理する能力を必要とするのに加えて、分散コンピューティングシステムのシステムソフトウェアに集中型システムにはない通信やセキュリティの課題を解決することが求められるためである。例えば、分散コンピューティングシステムの性能や信頼性

は利用する通信ネットワークの性能と信頼性にかなり依存する。一般に、特別なソフトウェアがユーザに対して性能や応答性を低下させるネットワークでのメッセージ伝送誤りの回復処理やネットワークの輻輳制御のために必要となる。同様に、アクセス制御やプライバシ保護に対して故意又は偶然の違反から広範囲に分散した共有される資源やサービスを守るためにセキュリティ度の測定が必要となる。

分散コンピューティングシステム構築における複雑性や困難性の増大にも拘わらず、分散コンピューティングシステムの導入や利用が急速に進んでいる。これは、分散コンピューティングシステムの利点がその欠点よりも優れているためである。分散コンピューティングシステムの出現や普及を促した技術的な要求、経済性及び主な利点を次に述べる。

1.4.1　本質的な分散アプリケーション

分散コンピューティングシステムは、必然的に出現してきたと考えられる。例えば、幾つかのアプリケーションはもともと分散されており、単純に実現すると分散コンピューティングシステムとなる。事例として、全国規模の従業員のデータベースを考えると、特定の従業員に関するデータはその所属する支店で作成され、全社的なデータベースとして扱えるとともに、各支店で作成されたデータはその場所で頻繁にかつ即座にアクセスすることが要求される。このようなアプリケーションは、各場所でデータの収集、前処理、アクセスを行うためにある程度の処理能力を要求し、その結果として分散コンピューティングシステムを必要とする。本質的な分散アプリケーションの事例として、全世界規模の飛行機の予約システム、利用者が各支店等の窓口で口座からお金の引き出しや預金を行う銀行システム、生産組み立てラインでのロボットや機械を制御するファクトリオートメーションシステムがある。

1.4.2　点在するユーザ間での情報の共有

分散コンピューティングシステムが出現した別の理由として、距離的に遠く離れた場所で情報の共有を行うための人と人との有効な通信手段が望まれた。分散コンピューティングシステムにおいては、ユーザの一人が生成した情報は容易にかつ効果的にそのシステムの別の場所で作業している人から共有することができる。この機能は非常に有効である。例えば、地理的に離れた人達の

コンピュータが同じ分散コンピューティングシステムに接続されていればプロジェクトを遂行することが可能となる。この場合、メンバーは地理的に離れているが、プロジェクトのファイルの転送、相互に遠隔ログインによるプログラムの実行、作業の調整を行うために電子メールを使った情報交換により協同作業ができる。

ユーザグループで協同作業を行うために分散コンピューティングシステムを使用するものとして、**コンピュータ支援協同作業 (computer-supported cooperative working)(CSCW)** や**グループウェア (groupware)** がある。グループウェアのアプリケーションは、分散コンピューティングシステムの別々のノードで実行するプログラム間でのデータオブジェクトの共有に深く関っている。グループウェアは、ソフトウェア開発者に対して大きな期待を与える新たな技術である。

1.4.3 資源共有

情報だけが、分散コンピューティングシステムにおいて共有できるものではない。プリンタ、ハードディスクやプロッタなどのハードウェア資源と同様ソフトウェアライブラリやデータベースなどのソフトウェア資源もまた、分散コンピューティングシステムのコンピュータやユーザの全てにおいて極めて効果的に共有できる。例えば、ワークステーションサーバモデルに基づく分散コンピューティングシステムにおいて、ワークステーションはディスクを持たないか、若しくは一時的な保管のために小容量なディスク (10–20 メガバイト) を持ち、全てのワークステーションからの大容量ディスク上の永続的なファイルへのアクセスは 1 つのファイルサーバによって提供される。

1.4.4 価格性能比の向上

これは、分散コンピューティングシステムの急速な普及のもっとも重要な要因の 1 つである。急速な処理能力の向上とミニコンピュータの低価格化、さらに通信ネットワークの高速化により、分散コンピューティングシステムは単一の大規模集中型システムに比べてより価格性能比が優れている。例えば、プロセッサプールモデルに基づく分散コンピューティングシステムでの小規模の CPU 数でも安価な端末から多くのユーザが効率良く利用でき、集中型のタイムシェアリングシステムやパーソナルコンピュータと比べてより価格性能比が高

いものであることがわかる。集中型システムに比べて分散コンピューティングシステムがより価格性能比が良い理由は、複数のコンピュータ間での資源共有機構を提供している点にある。例えば、カラーレーザプリンタ、高速記憶装置、プロッタなどの高価な周辺装置それぞれが同じ分散コンピューティングシステムにおいて全てのコンピュータから共有することができる。もし、コンピュータが通信ネットワークによって相互に接続していない場合、各コンピュータはそれぞれが周辺装置を持つことになり、結果的に高価なシステムになる。

1.4.5　短い応答時間と高スループット

複数のプロセッサの導入により、分散コンピューティングシステムは単一プロセッサ構成の集中型システムに比べて高性能が期待できる。もっとも標準的に使用される評価尺度は、ユーザのプロセスの応答時間とスループットの 2 つである。すなわち、分散コンピューティングシステムの複数のプロセッサは、単一プロセッサ構成の集中型システムよりも短い応答時間と高スループットを実現するために有効である。例えば、2 つの別々なプログラムを実行する場合、2 つのプロセッサの方が別のプロセッサ上で同時に実行ができるため 1 つのプロセッサよりも強力に処理できる。さらに、ある処理を行う場合独立に実行できるような処理単位に分割することができるならば、分散コンピューティングシステムではそれぞれの処理単位を各プロセッサで同時に実行することができる。高速な通信ネットワークによる分散コンピューティングシステムは、複雑な問題を即座に処理するための並列コンピュータとしてしだいに利用されはじめている。全体のスループットを向上するための分散コンピューティングシステムの別の利用方法として、過負荷状態のプロセッサのジョブを余裕のあるプロセッサに移動させることで複数のプロセッサ間の負荷分散を実現する方法がある。例えば、ワークステーションモデルに基づく分散コンピューティングシステムにおいて、ユーザが 2 つの実行すべきプロセス、1 つは相互動作を行うもので他の 1 つはバックグランドで実行できるプロセスをもつ場合、そのユーザのホームノードで相互作用を行うプロセスを実行し、他のプロセスは (空きのノードがあれば) 遠隔の空きのノードで実行することによってより効果的となる。

1.4.6 高信頼性

信頼性 **(reliability)** は、システムにおける誤りや構成機器の障害に対する耐久度である [Stankovic 1984]。信頼性のあるシステムは、構成機器の障害に対しても情報の紛失を防ぐことができる。分散コンピューティングシステムの記憶装置やプロセッサの重複は、システム内での重要な情報の複製保持や重要な処理の実行を冗長的処理することにより対応できる。この方法により破壊的な障害に対してシステムを守ることを可能している、プロセッサの1つに障害が発生しても他のプロセッサによって処理を最後まで実行でき、また記憶装置の1つが故障しても別の記憶装置の情報が使用できる。さらに、分散コンピューティングシステムのプロセッサや他の資源が地理的に点在していることによって自然災害がおよぼす障害の範囲を抑えることができる。

信頼性の重要な点としてシステムが使用できる時間を表す**可用性 (availability)** がある。集中型システムに比べて、分散コンピューティングシステムは可用性の向上が可能である。例えば、集中型システムにおいてプロセッサが障害を起こした場合 (単一プロセッサ構成の集中型システムを想定する)、システム全体がダウンし、処理ができなくなってしまう。しかし、分散コンピューティングシステムではシステムの1部が他の部分を使っているユーザのジョブた影響なくダウンする。例えば、ワークステーションサーバモデルに基づく分散コンピューティングシステムの1台のワークステーションが障害になっても、そのワークステーションのユーザのみに影響し、他のユーザはこの障害の影響を受けない。同様に、プロセッサプールモデルに基づく分散コンピューティングシステムにおいて、プールのあるプロセッサが障害となってもシステムはダウンしたプロセッサの数に比例して性能は低下するが正常に動作しつづける。この場合には、ユーザは直接影響を受けることがなく、また障害が発生したことにも気付かない。障害が致命的となる重大なアプリケーションにとっては高信頼性の観点からも分散コンピューティングシステムは重要である。しかし、信頼性はその価格に関係してくる、したがって、この2つの関係のバランスをいかに保つかが重要になる。

1.4.7 規模拡張性

分散コンピューティングシステムの別の長所として規模拡張性がある。すなわち、必要になった時に簡単に資源 (ハードウェアとソフトウェア) を追加する

ことによって、分散コンピューティングシステムの処理能力や機能を段階的に高めることができる。例えば、処理負荷の増加に対してプロセッサをシステムに追加することで容易に対応でき、規模の拡張ができる。このような規模拡張性は、多くの既存のアプリケーションやこれから現れるアプリケーションにとって将来どの程度のシステム処理能力が必要となるかは予測し難いため、大変に魅力ある機能である。分散コンピューティングシステムにおいては、システムの機能構造を変えることなく既存のシステムに新たな資源を追加できるためにその規模拡張性は容易に実現できる。規模拡張性を持つように設計されている分散コンピューティングシステムは、**オープン分散システム (open distributed systems)** と呼ばれる。

1.4.8　ユーザ要求への融通性

一般に、各種のタイプのコンピュータがあればそれぞれに適した処理を実行することができる。例えば、普通の処理能力を持つコンピュータは普通のデータ処理に向いており、高速なコンピュータは複雑な計算処理に適している。集中型システムでは、ただ1台のコンピュータで各種のタイプの処理を行わなければならない。しかし、分散コンピューティングシステムでは各種のコンピュータをプールとして持っており、ユーザのジョブに適したコンピュータを選択することができる。すなわち、ハイブリッドモデルに基づく分散コンピューティングシステムでは相互作用を行うジョブはユーザのワークステーションで実行し、その他のジョブについてはプールされているプロセッサで処理することができる。

上述した分散コンピューティングシステムの特長は自然に得られるものではなく、分散コンピューティングシステムを十分考慮しながら設計することで実現できる。本書は、これらの特長を実際に達成するための各種の設計方式について述べる。

1.5　分散オペレーティングシステムとは

Tanenbaum や Van Renesse[1985] は、**オペレーティングシステム (operating system)** をコンピュータシステムの資源を制御し、裸のマシンよりも使い勝手の良いインタフェースや**仮想マシン (virtual machine)** をユーザに提供するプログラムとして定義している。この定義にしたがうと、オペレーティング

システムの主要な機能は次の 2 つである。

1. 直接ハードウェアを意識しないで容易にプログラムできる仮想マシンをユーザに提供すること。
2. システム内の各種の資源を管理すること。すなわち、資源の利用状況の把握と割り当て、プログラムやユーザからの資源の利用要求の管理と制御などを行うことである。

したがって、コンピュータシステムのユーザへの見え方、コンピュータシステムの各種資源へのユーザのアクセス方法及び資源への要求の取り扱い方は、コンピュータシステムのオペレーティングシステムに大いに依存する。分散コンピューティングシステムで一般に使用されるオペレーティングシステムは、**ネットワークオペレーティングシステム (network operating systems)** と **分散オペレーティングシステム (distributed operating systems)** の 2 つに大きく分類できる。これらの 2 つのオペレーティングシステムの相違を表す 3 つの重要な機能として、システムのイメージ、自律性、フォールトトレラント性がある。これらの機能について以下に述べる。

1. **システムイメージ (system image)** :これらの 2 つのオペレーティングシステムの相違を表す最も重要なものとして、ユーザから見える分散コンピューティングシステムのイメージである。ネットワークオペレーティングシステムの場合では、通信ネットワークによって接続された異なった複数のマシンの集合として分散コンピューティングシステムをとらえる。すなわち、ユーザは複数のコンピュータの存在を意識する。一方、分散オペレーティングシステムでは複数のコンピュータの存在を隠蔽し、ユーザは単一のシステムイメージとしてとらえる。すなわち、ネットワークにつながったマシンの集合を**仮想ユニプロセッサ (virtual uniprocessor)** としてみなす。この機能に基づく 2 つのオペレーティングシステムの違いを理解するために、2 つの事例を述べる。

ネットワークオペレーティングシステムの場合、ユーザは分散コンピューティングシステムのどのマシンでもジョブを実行することができるが、ジョブをどこで実行するかは意識することになる。すなわち、ユーザがログインしているマシンでジョブを実行することが基本である。別のマシンで実行するためには「遠隔ログイン」コマンドか、若しくは遠隔のマシンで実行するための特殊なコマンドを使って遠隔のマシンにログインしなければならない。どちらの

場合でも、ユーザがジョブを実行するマシンを知ることになる。一方、分散オペレーティングシステムでは処理を行うためにシステムのマシンにジョブを動的にかつ自動的に割り付ける。したがって、分散オペレーティングシステムのユーザは、一般にジョブが実行しているマシンを知らない。すなわち、ジョブを実行するマシンの選択は、ネットワークオペレーティングシステムでは手動であり、分散オペレーティングシステムでは自動的行われる。

ネットワークオペレーティングシステムでは、ユーザはアクセスする資源の位置を知っている必要があり、またローカルと遠隔の資源をアクセスするために別のシステム呼び出しを使用しなければならない。一方、分散オペレーティングシステムではアクセスするためにそれらの位置を気にする必要がなく、ローカルと遠隔の資源をアクセスするために同じシステム呼び出しを使用できる。したがって、ネットワークオペレーティングシステムでは通常ユーザはファイルがどこにあるかを意識し、ファイルをあるマシンから別のマシンに転送する場合には**ファイル転送 (file transfer)** のコマンドを使用しなければならない。しかし、分散オペレーティングシステムではローカルか遠隔かのファイルの位置を気にしないでファイルへのアクセスのために同じコマンドを使用できる。すなわち、ネットワークオペレーティングシステムではファイル位置に関する制御はユーザが行うのに対して、分散オペレーティングシステムではシステムが自動的に制御する。

この機能は、基本概念である「**透過性 (transparency)**」に基づいている。本章の後半では、分散オペレーティングシステムがユーザに単一のシステムイメージを提供するためにいろいろな透過性の機能を持っていることを述べる。しかし、現状の分散オペレーティングシステムの機能では十分にこの目標を達成しておらず、今後の研究活動が期待されている。

2. **自律性 (autonomy)**:ネットワークオペレーティングシステムは、既存の集中型システムの集合として構築され、遠隔操作のインタフェースや調整及びこれらのオペレーティングシステム間の通信を制御する。すなわち、ネットワークオペレーティングシステムでは分散コンピューティングシステムの各コンピュータはそれぞれローカルのオペレーティングシステム (異なるコンピュータのオペレーティングシステムは同じものもあれば違う場合もある) を持っており、異なるコンピュータの 2 つのプロセスが相互に通信するための規則、通信プロトコルを除けばコンピュータ相互の調整は必要ない。したがって、各コンピュータはそこでのプロセスの生成や終了及びローカルの資源管理に関して

独自に制御し、他のコンピュータとは独立している。ローカルのオペレーティングシステムの相違を考慮して、別のコンピュータの分散オペレーティングシステムが同じでもシステム呼び出しは異なるものとこの場合は考える。

一方、分散オペレーティングシステムでは個々のシステムを越えた1つのオペレーティングシステムが存在しており、分散コンピューティングシステムの各コンピュータはこの全体的なオペレーティングシステムの1部として動作する。分散オペレーティングシステムでは、システムの各種の資源の利用率を高めるために分散コンピューティングシステムの全てのコンピュータは密に相互動作を行う。すなわち、プロセスや資源は全体的に管理される(一部の資源はローカルで管理される)。さらに、分散コンピューティングシステムの全てのコンピュータに対して使用できるシステム呼び出しのセットが用意される。

オペレーティングシステムがサポートするシステム呼び出しのセットは、オペレーティングシステムの**カーネル (kernel)** と呼ばれるプログラムとして実装される。カーネルは、システム呼び出しを介して他のプログラムからアクセスされる機能や資源を提供するためにコンピュータシステムのハードウェアを管理、制御する。システム全体で使用できるシステム呼び出しのセットを実現するために、分散オペレーティングシステムではカーネルは分散コンピューティングシステムの全てのコンピュータで実行する。異なるコンピュータ上のカーネルは、全体としての動作判断を行うために、例えばシステムに新たに生成されたプロセスを最も最適に実行できるマシンを見つけ出すために相互動作を行う。

すなわち、ネットワークオペレーティングシステムでのマシンの自律性は分散オペレーティングシステムのマシンの自律性に比べて高いと言える。

3. **フォールトトレラント能力 (fault tolerance capability)**:ネットワークオペレーティングシステムは、全体の分散コンピューティングシステムの10 %のマシンがダウンすると少なくともユーザの10 %は作業を継続できなくなるという点ではほとんどフォールトトレラント能力を持っていない。一方、分散オペレーティングシステムではマシンの障害の影響をほとんどのユーザは通常受けることなく、分散コンピューティングシステムシステムの性能を10 %低下するのみで作業を継続することができる。したがって、分散オペレーティングシステムのフォールトトレラント能力はネットワークオペレーティングシステムと比べて高い。

Tanenbaum と Van Renesse[1985] が与えている次のような分散オペレーティ

ングシステムの定義は、これまで述べたような機能のほとんどを含んでいる。

分散オペレーティングシステムは従来の集中型オペレーティングシステムのようにユーザには見えるが、独立した複数の集中処理装置 (CPU) で実行する。ここでの基本概念は**透過性 (transparency)** である。言い替えれば、複数のプロセッサが使用されていることはユーザは意識しない (透過性)。この考え方のもう 1 つの表現はユーザはこのシステムを異なるマシンの集合としてではなく「**仮想ユニプロセッサ (virtual uniprocessor)**」として見るということである。

ネットワークオペレーティングシステムを使用する分散コンピューティングシステムは通常**ネットワークシステム (network system)** と呼ばれ、分散オペレーティングシステムを使用するものを通常真の分散システム (又は単に**分散システム (distributed system)**) と呼ぶ。本書では真の分散システムとして分散システムの用語を使用する。現在の分散オペレーティングシステムの技術では完全な真の分散システムを設計することはできていないため、これを実現するすることが分散オペレーティングシステム分野の研究者の究極の目的となっている。

1.6　分散オペレーティングシステム設計の課題

一般に、**分散オペレーティングシステム (distributed operating system)** の設計は幾つかの理由から集中型オペレーティングシステムの設計よりも難しい。集中型オペレーティングシステムでは、オペレーティングシステムが機能する環境として完全にかつ正確に情報のアクセスできることが前提になっている。例えば、集中型オペレーティングシステムで状態の問い合わせを行う場合、コマンドによって問い合わせするのは単一のオペレーティングシステムのみであり、かつ問い合わせ中に状態を変化させることがないため問い合わせの結果は正しく戻される。しかしながら、分散オペレーティングシステムではシステム環境に関する情報が完全に与えられないことを前提にして設計しなければならない。分散システムでは、資源は物理的に分散しており、複数のプロセッサに対して共通の時刻がなかったり、メッセージの配信で遅れが生じたり、またメッセージの紛失が発生することもある。このような理由から、分散オペレーティングシステムでは分散システムにおける各種の構成要素の状態について最新の一貫性のある情報を持てない。したがって、最新の一貫性のある情報の欠如はいろいろな面で (例えば、資源の管理や相互動作での同期) 分散オペレーティングシステムの設計をより困難にしていることは明らかである。例え

ば、オペレーティングシステムがプロセッサの状況を逐一管理できなければプロセッサのスケジュールを管理するは困難となる。

これらの複雑性や困難性にかかわらず、分散オペレーティングシステムは分散システムの長所をユーザに提供するために設計されなければならない。ユーザにとっては**分散システム (distributed system)** は仮想集中型システムとして見える。すなわち柔軟性があり、効果的で、信頼性に富み、安全で使い勝手の良いことが要求される。この要求に応えるために分散オペレーティングシステムの設計者は幾つかの設計課題に取り組まなければならない。幾つかの主要な設計課題を以下に述べるとともに、本書では設計課題の詳細な説明と共通に使用される技術について説明する。

1.6.1 透過性

分散オペレーティングシステムの主要な目標は、複数のコンピュータの存在を隠蔽し (**透過性 (transparency)**)、ユーザに単一システムのイメージを提供することである。すなわち、分散オペレーティングシステムは、通信サブシステムによって接続された各種のマシンの集合体が**仮想ユニプロセッサ (virtual uniprocessor)** としてユーザに見えるように設計されなければならない。完全な透過性を達成することは、困難が伴い、分散オペレーティングシステムが持つべき幾つかの異なった透過性が要求される。開放型分散処理に対する国際標準化機構の参照モデル (**reference model**)[ISO 1992] では、8 種類の透過性、すなわちアクセス透過性、位置透過性、複製透過性、故障透過性、移動透過性、並行透過性、性能透過性及び規模透過性が示されている。これらの透過性の機能について以下に述べる。

アクセス透過性

アクセス透過性 (access transparency) とは、資源 (ハードウェア又はソフトウェア) が遠隔又はローカルのどこにあるかをユーザが意識しなくてよい、又は意識できないことを意味している。このことは、ユーザがローカルの資源にアクセスするのと同じ方法で遠隔の資源にアクセスできることを分散オペレーティングシステムが可能にしていることを示している。すなわち、システム呼び出しの形式で提供されるユーザインタフェースはローカルと遠隔の間には相違がなく、ユーザに対して透過性のある方法でユーザからの要求を資源の位置に関わらず分散オペレーティングシステムの責任の元で制御する。

このような要望は、集中型や分散型の環境においてもまた全体的な資源の名前付け機能においても重要であり、十分に設計されたシステム呼び出しのセットが必要となる。完全なアクセス透過性を提供するシステム呼び出しを設計することは不可能に近いが、第3章及び第4章で分散システムにおける通信の障害に対応するための要求に関して説明する。一方、全体的な資源の名前付け機能の設計に関してはこれまで多くの研究が行われている。第10章に全体的な資源の名前付け機能の基本概念と設計法について述べる。第5章で述べている分散共用メモリ機構は、ローカルと遠隔のメモリオブジックトの両者をマクセするための統一的なシステム呼び出しのセットを与えるものと考えることもできる。この機構はアクセス透過性を提供するために有効であるが、性能の制約があるため限られたタイプの分散アプリケーションに適用される。

位置透過性

位置透過性 (location transparency) は次の2つの主な特徴を持つ。

1. **名前透過性 (name transparency)** :これは、資源 (ハードウェアやソフトウェア) の名前がその資源の物理的位置に関する情報を何ら持たないことを意味する。すなわち、資源の名前はシステムの物理的な接続性や構成法又は資源の位置とは独立である。さらに、名前の変更は生じないで資源 (例えばファイル) が分散システム内のあるノードから別のノードに移動しなければならない。したがって、資源の名前はシステム内で一意となる。
2. **ユーザ可動性 (user mobility)**:これは、ユーザがログインしたマシンに依存しないで同じ名前で資源にアクセスできることを意味する。すなわち、ユーザはシステム内の異なるノードから同じ資源をアクセスする場合に別々の名前を必要とされない。ユーザ可動性を提供する分散システムでは、ユーザはシステムのどのマシンにも自由にログインでき、何ら気にすることもなく資源にアクセスできる。名前透過性とユーザ可動性の両者ともシステム全体にわたった資源名前付け機能と関連する。

複製透過性

性能や信頼性の向上のために、ほとんど全ての分散オペレーティングシステムは分散システムのノードにあるファイルや他の資源の複製 (付加的なコピー) を行う。このようなシステムでは、複製された資源の存在やその状況はユーザに対して透過的でなければならない。すなわち、**複製透過性 (replication**

transparency) に関する 2 つの重要な課題、複製の名前付けと複製の制御がある。資源の複製に対する名前付け及びユーザが与えた名前と複製した資源との対応付けはシステムの責任範囲である。さらに、複製の制御方法、例えば複製の数、複製の置き場所、複製の生成と消滅の時期についてはユーザに意識させない方法でシステムが自動的に行わなければならない。複製の管理上の課題については第 9 章で述べる。

故障透過性

故障透過性 (failure transparency) は、システムにおける部分的な故障、例えば、通信回線障害、マシン故障又は記憶装置のクラッシュを隠蔽することである。故障透過性の機能を持つ分散オペレーティングシステムは、部分的な故障が発生しても能力を下げながら機能を継続する。例えば、分散オペレーティングシステムのファイルサービスでの故障透過性を考えてみよう。これは、システムのファイルを管理するためにお互いが密に協調して行い、またユーザはあたかも 1 台のファイルサーバが動作しているようにファイルサーバを利用できるようにファイルサーバのグループとして実現することである。この場合、ユーザはファイルアクセス動作が遅くなることがあるものの、1 台若しくは数台のファイルサーバの故障に気付くことはない。このような方法であらゆるタイプのサービスに対して故障透過性を実現することができる。しかし、複数のサーバが相互に連携するために生じるシステムのオーバヘッドを抑制することが設計上必要となる。

完全な故障透過性は、全ての障害をユーザに隠蔽することができないため現在の分散オペレーティングシステムおいては実現できない。例えば、分散システムの通信ネットワーク障害はユーザの仕事を中断させ、ユーザからの苦情が来る。さらに、完全な故障透過性に対応する分散システムを設計することは、あらゆるタイプの故障に耐えられるようにするために大掛かりな冗長システムとなるため低速で高価なシステムになってしまう。したがって、このような分散システムを設計することは理論的には可能であるが実際的ではない。

移動透過性

性能の向上、信頼性及びセキュリティの面から、移動できるオブジェクト (例えば、プロセスやファイル) は分散システムのあるノードから他のノードにしばしば移動する。**移動透過性 (migration transparency)** の目的は、オブジェクトの移動をユーザとは無関係にシステムによって自動的に行うことであ

る。この目標を達成するための課題として次の 3 つがある。

1. オブジェクトがどこからどこへ移動するかの決定はシステムが自動的に行う。

2. あるノードから他にオブジェクトが移動することによってその名前が変わらない。

3. 移動するオブジェクトがプロセスの場合、移動するプロセス宛のメッセージが受信される前にプロセスが移動しても送信側のプロセスが意識することなくプロセスに届くためのプロセス間通信機構がある。

第 7 章で 1 番目の課題を述べる。2 番目の課題は全体的な資源の名前付け機能に関係し、第 10 章で記述する。3 番目の課題については第 8 章で説明する。

並行透過性

分散システムでは、空間的に離れている複数のユーザが同時にシステムを利用する。このような状況では、同時に実行するユーザのプロセス内でシステムの資源 (ハードウェアやソフトウェア) を共有することは経済的である。しかしながら、コンピューティングシステムで利用可能な資源の数が制限されているため、1 つのユーザのプロセスは資源の競合のために同時に実行する他のユーザプロセスの行動に何らかの影響を与える。例えば、2 つのプロセスが同じファイルを同時に更新する場合を防ぐ必要がある。**並行透過性 (concurrency transparency)** は、各ユーザは自分がシステムの唯一のユーザであり、他のユーザはシステムに存在しないと感じることを意味する。並列透過性を提供するために、分散オペレーティングシステムの資源共有機構として次の 4 つの機能を持たなければならない。

1. 事象順序性機能は、各種のシステム資源への全てのアクセス要求がシステムの全てのユーザに一貫性を提供するように適切に処理されることを保証する。

2. 相互排他機能は、いかなる時もプログラム動作が正しいならば複数のプロセスが同時に 1 つの資源にアクセスしないように 1 つのプロセスが共用資源にアクセスすることを保証する。

3. 非飢餓機能は、複数のプロセスが同時にアクセスしないように管理される資源を利用できる全てのプロセスが資源を解放すれば、その資源への要求は受け入れられることを保証する。

4. 非デッドロック機能は、1つのプロセスがシステムで利用可能な資源以上のものを要求しないとしても、競合するプロセスが相互に要求を牽制しあうような状況を発生しないことを保証する。

第6章では上記の並行透過性の課題について述べる。

性能透過性

性能透過性 (performance transparency) の目的は、システムで動的に変化する負荷に応じて性能を最適にするために自動的にシステムの再構成を行うことである。実際には、システムの1つのプロセスに負荷がかかり、他のプロセスがアイドル状態になることを発生しないことである。すなわち、システムの処理能力をその時点でのシステムのジョブに均等に分割することである。

このためには、分散オペレーティングシステムにおいて知的な資源配置機能とプロセス移動機能が必要となる。第7章と第8章はこれらの課題について述べる。

規模透過性

規模透過性 (scaling transparency) の目的は、ユーザの活動を中断することなくシステムの規模を拡張できることである。このためには、分散オペレーティングシステム構成要素の設計におけるオープンシステムのアーキテクチャと拡張性を持つアルゴリズムの利用が必要となる。本章の1.6.3と第2章の2.6でオープン分散システムの設計の課題について述べる。一方、分散オペレーティングシステムの構成要素は拡張性を持つアルゴリズムを利用しなければならないため、この課題については本書のあらゆる箇所で述べる。

1.6.2　信頼性

一般に、分散システムは複数の資源のインスタンスが存在するため集中型システムよりも**信頼性 (reliability)** が要求される。しかし、複数の資源のインスタンスの存在はシステムの信頼性を高めるものではない。むしろ、これらの資源を管理する分散オペレーティングシステムは、この分散システムの特長を十分に活かしてシステムの信頼性を高めるために設計されなければならない。

障害 (fault) は、誤りを生じさせる機械的又はアルゴリズム的な欠陥である。システムの障害は、システムの故障を生じる。故障をしたシステムの動作方法に依存して、システム故障は**停止型障害 (fail-stop)**[Schlichting and Schneider 1983] と**ビザンチン障害 (Byzantine failure)** [Lamport et al. 1982] の 2 種類がある。停止型障害の場合、システムは故障が検出できる状態に遷移した後機能を停止する。一方、ビザンチン障害の場合システムは機能し続け、間違った結果を生じる。検出不可能なソフトウェアのバグは、しばしばシステムのビザンチン障害を生じる。確かに、ビザンチン障害の方が停止型障害より扱い方がより困難である。

高信頼性のために、分散オペレーティングシステムの障害処理機構は、障害を避け、障害に耐えうるためや障害の検出と復旧のために適切に設計されなければならない。これらの課題に関する共通的に使用される方法についてその概要を以下に示す。

障害回避

障害回避 (fault avoidance) は、障害の発生を最小限にするためのシステム構成要素の設計扱う。従来の設計法、例えば高信頼な設計法を用いる方法が障害回避の考え方に基づいてシステムの信頼性を高めるためにしばしば使用される。分散オペレーティングシステムはハードウェア構成要素の障害回避能力に対してほとんど寄与しないため、分散オペレーティングシステムの各種のソフトウェア構成要素の設計者はこれらの構成要素の信頼性を高めるために十分に試験を行わなければならない。

フォールトトレラント性

フォールトトレラント性 (fault tolerance) は、システムの一部の故障が発生してもシステムとして機能を継続することができる能力である。システムの性能は、部分的な故障のために低下するかもしれないが、システムとして適切に機能し得る。分散オペレーティングシステムのフォールトトレラント性の能力を高めるための重要な考え方を下記に示す。

1. **冗長性技術 (redundancy techniques)**。冗長性技術の基本的な考え方は、構成要素の 1 つが故障しても他のもので対応できるように重要なハードウェアやソフトウェアの構成要素を置き換えできるようにし、故障の単一箇所性を避けることである。明らかに、重要な構成要素に対しては 2、3 の予備を

持つことにより、少なくとも原則として、部分的な故障が発生してもオペレーションを継続することはできる。例えば、重要なプロセスは 2 つのノードで実行することで、一方のノードで障害が起きても他のノードでプロセスの実行は完了することができる。同様に、重要なファイルは信頼性を高めるために複数の記憶装置に置くことが考えられる。

冗長性技術を導入することで、資源の複製を保守したり、またそれらの一貫性を保つために新たなシステムオーバヘッドが生じることに注意することが必要である。例えば、1 つのファイルが分散システムの複数のノードに複製されると、余分にディスク容量が必要となり、正しく機能するためにファイルの全ての複製に相互の一貫性が要求される。一般に、複製の数を大きくすれば信頼性は高まるが、システムオーバヘッドは大きくなる。したがって、分散オペレーティングシステムは信頼性とオーバヘッドのバランスを適切に保つように設計することが必要である。この点は重要な問題であり、どの程度の複製の数であれば十分か？ この問題に対する解は、k 個の構成要素の故障に対しても機能を継続できる **k 重フォールトトレラント (k-fault tolerant)** の問題に帰着する [Cristian 1991Nelson 1990]。したがって、システムを k 重停止型障害に耐えられるように設計するためには、k+1 個の複製が必要になる。k 個の複製が障害となったとしても残りの 1 つによってシステム機能を継続することができる。一方、システムを k 重ビザンチン障害に耐えられるように設計するためには、少なくとも 2k+1 個の複製が必要になる。これは、k 個の複製が異常になった場合 k+1 個の複製を多数として採用する多数決の方法に基づくためである。

メモリオブジェクトの複製と一貫性制御機構は第 5 章に、ファイルオブジェクトに関しては第 9 章で述べる。

冗長性技術の他の応用は、安定記憶装置の設計があり、一時的な I/O 障害や記憶媒体の寿命に対処できる仮想記憶装置がある。重要なファイルの信頼性は、安定記憶装置に格納することで改善される。安定記憶装置について第 9 章で述べる。

2. **分散制御 (distributed control)**。信頼性を高めるために、分散オペレーティングシステムで使用される多くの特殊なアルゴリズムやプロトコルが故障の単一箇所性を避けるための分散制御機構として動作する。例えば、高稼動な分散ファイルシステムは複数の独立した記憶装置を制御する複数の独立したファイルサーバを持つべきである。ファイルサーバに加えて、分散制御技術は名前サーバやスケジューリングアルゴリズム及び他の実行制御機能にも利用さ

れる。複数の分散サーバが特殊なサービスを提供するために分散システムで使用される場合、サーバが非依存であることが重要である。すなわち、設計として同じ様な機能のサーバを要求していない。さもなければ、信頼性は悪化する。分散制御機構については本書全般で言及する。

障害検出と復旧

信頼性向上のための障害検出と復旧方法は、故障の発生を検出し、オペレーションを継続できる状態にシステムを正常にするためのハードウェアとソフトウェア機構の利用に関係する。分散オペレーティングシステムにおいてこの方法を実現するために共通的に使用される技術は、以下のものがある。

1. **原子トランザクション (atomic transactions)**:原子トランザクション (単に**トランザクション (transaction)**) は、障害や並列処理において個別に行われる動作の集合からなる計算処理である。すなわち、全てのオペレーションが成功裡に遂行されるか、うまくいかないかのどちらかであり、また並行に実行する他のプロセスは中間的な状態を変更又は観ることはできない。トランザクションは、障害や同時アクセスにおいて共有されるデータオブジェクト (例えばファイル) の集合の一貫性を保証するための助けとなる。トランザクションは、全てのオペレーションが成功裡に遂行されるか又は全く遂行されないかの 2 つの状態をとるだけであるため、クラッシュからの回復を容易にする。

トランザクション機能を持つシステムでは、トランザクションが完了する前にハードウェア障害若しくはソフトウェアの誤りによってプロセスが思わぬ停止を行った場合、システムは最初の状態に移行するためにデータオブジェクトを待避する。もし、システムがトランザクション機構を備えていない場合、オペレーションの処理中に期待しないプロセスの障害が発生するとデータオブジェクトは一貫性のない状態に置かれしまう。したがって、トランザクション機能を持たないならば、一貫性のない状態から最初の状態にデータオブジェクトを回復することは困難であり、不可能に近い。原子トランザクションについては第 9 章で述べる。

2. **ステートレスサーバ (stateless servers)**:クライアントサーバモデルは、ユーザの要求を処理するためにしばしば分散システムで使用される。このモデルにおいては、サーバは次の 2 つのパラダイム、状態保持 (ステートフル) 若しくは状態不保持 (ステートレス) のどちらかによって実現される。2 つのパラダ

イムは、クライアントとサーバの関係の特徴によって区別される。すなわち、クライアントとサーバ間のサービスされる要求の履歴が次のサービス要求に影響するか否かによる。状態保持の方法はサービスされる要求の履歴に依存するが、状態不保持の方法は履歴に依存しない。ステートレスサーバは、障害発生に対して**ステートフルサーバ (stateful server)** とは異なった利点を持っている。すなわち、ステートレスサービスパラダイムは、クライアントの状態情報をサーバが保守しないためクラッシュの回復を大変容易に行うことができる。一方、ステートフルサービスパラダイムは複雑なクラッシュの回復手順を必要とする。クライアントとサーバとも正しくクラッシュの検出を行う必要がある。サーバはクライアントのクラッシュを検知し、必要な誤り処理を行う必要がある。場合によってステートフルサービスが、障害検出と回復処理を単純化するためにが必要となることもある。可能な限りステートレスサービスパラダイムを使用すべきである。ステートレス及びステートフルサービスについては、第4章と第9章で述べる。

3. 応答とタイムアウトによるメッセージの再送:分散システムにおいて、ノードのクラッシュや通信回線の障害が2つのプロセス間の通信中に発生すると、メッセージが欠落することがある。したがって、信頼性のある**プロセス間通信機構 (interprocess communication mechanism)** はメッセージが欠落したことを検出し、再送できなければならない。メッセージの欠落に対する処理は、通常、メッセージの応答とタイムアウトによる再送を含む。すなわち、受信側は受信した全てのメッセージに対して応答メッセージを返送しなければならない。送信側がある一定時間内に応答メッセージを受けなければ、メッセージが届かなかったものとしてメッセージの再送を行う。この方法の問題点は、メッセージの重複である。重複するメッセージが障害やタイムアウトのために送られる。したがって、信頼性のあるプロセス間通信機構は重複メッセージの検出と処理ができなければならない。重複メッセージの処理には、一般的に自動的に順序番号を生成し、メッセージに付与する機構が含まれる。信頼性を高めるための応答メッセージの使用、タイムアウトによる再送及び重複メッセージの処理については第3章で記述する。

上述した機構は、信頼性の高い分散システムを構築するために有効である。しかしながら、システムの信頼性向上における主な弱点はこれらの技術による新たなオーバヘッドが実行時間に影響を与える点にある。多くのシステムにお

いて、数多くの信頼性を高めるための機構を導入することは、コスト高になる。

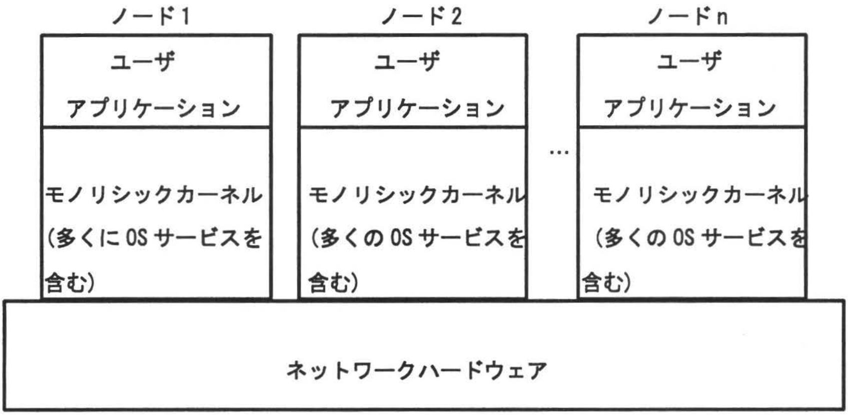
1.6.3　柔軟性

分散オペレーティングシステムの設計における別の重要な課題は、**柔軟性 (flexibility)** である。柔軟性は、オープンな分散システムの最も重要な機能である。分散オペレーティングシステムの設計は、次の理由から柔軟性を持つべきである。

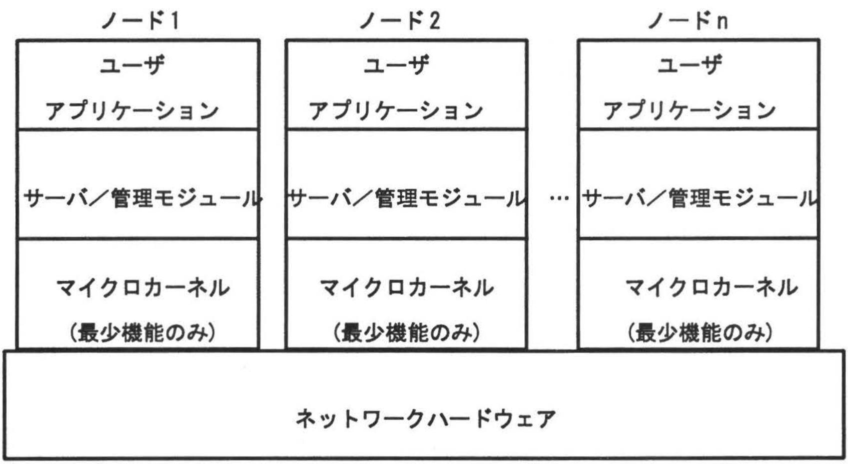
1. **変更の容易性 (ease of modification)** . システム設計者の経験から、設計上の不具合やシステム環境の変化や新たなユーザ要求に適用できない設計となるためしばしば設計のある部分は置き換えや変更が必要となる。したがって、ユーザに影響を与えないか、若しくは影響を最小限にするためにシステム内の変更を容易に取り込むことが必要である。
2. **拡張の容易性 (ease of enhancement)**. 全てのシステムにおいて、日々新しい機能が性能向上や使い勝手の向上のために追加される。したがって、システムに新たなサービスを容易に追加できることが望まれる。さらに、あるユーザのグループがオペレーティングシステムによって特定のサービスが提供される形態を好まない場合、そのユーザのグループが親しみがあり、かつ快適に感じるような形態で作業ができるサービスを追加して使用できるような柔軟性が必要である。

分散オペレーティングシステムの柔軟性に関わる最も重要な設計要素は、その**カーネル (kernel)** を設計するために使用するモデルである。オペレーティングシステムのカーネルは、基本的なシステム機能を提供する中央制御部である。それはユーザのプロセスがアクセスできない別のアドレス空間で動作し、ユーザは置き換えや変更することができない部分である。分散オペレーティングシステムでは、同じカーネルが分散システムの全てのノードで動作することを既に説明した。

分散オペレーティングシステムのカーネル設計で共通的に使用される 2 つのモデルは、**モノリシックカーネル (monolithic kernel)** と**マイクロカーネル (microkernel)** である (図 1.6)。モノリシックカーネルモデルでは、多くのオペレーティングシステムサービス、例えばプロセス管理、メモリ管理、装置管理、ファイル管理、名前管理及びプロセス間通信はカーネルによって提供され



(a) モノリシックカーネルモデル、カーネルレベルの上位はユーザによって階層化することができる。



(b) マイクロカーネルモデル、この図ではカーネルレベルの上位を2つのレベルとして示しているが、ユーザは多階層化することができる。

図 **1.6**　分散オペレーティングシステムのカーネル設計モデル

る。そのため、カーネルは大きく、一枚岩の構造になる。UNIX オペレーティングシステムを拡張したものや類似のものの分散オペレーティングシステムの多くは、モノリシックカーネルモデルを使用している。これは、UNIX 自身が大きく、一枚岩のカーネルであることが主な要因である。

一方、マイクロカーネルモデルはその目的が可能な限りカーネルを小さくすることにある。そのため、このモデルのカーネルは付加的なオペレーティングシステムサービスを実現するために必要となる最小機能のみを提供する大変小さいソフトウェアの核である。このモデルのカーネルが提供するサービスは、プロセス間通信、低レベルな装置管理、限られた低レベルなプロセス管理及び限定したメモリ管理である。ファイル管理、名前管理、付加的なプロセス及びメモリ管理などのオペレーティングシステムサービスや多くのシステム呼び出し処理は、ユーザレベルのサーバプロセスとして実現される。各サーバプロセスは、それ自身のアドレス空間を持ち、別々にプログラムすることができる。 モノリシックカーネルモデルと比較して、マイクロカーネルモデルは幾つかの利点がある。モノリシックカーネルモデルでは、カーネルが大きいためオペレーティングシステムとして柔軟性や構成法に難点がある。一方、マイクロカーネルモデルによるオペレーティングシステムは本質的に高いモジュール性を持っている。この特徴的な機能によりマイクロカーネルのオペレーティングシステムは、設計、実現及び実装が容易である。さらに、サービスの多くがユーザレベルのサーバプロセスとして実現されるため、設計の変更や新たなサービスの追加が容易である。これはまたオペレーティングシステムが提供する特定のサービスを好まないユーザに対してユーザ自身のサービスを実現することを可能にする。さらに、サービスの追加や変更においてモノリシックカーネルのようにシステムを停止して新たなカーネルを起動する必要はない。しかしながら、マイクロカーネルモデルに基づいて設計する場合、性能上の課題がある。これは、マイクロカーネルモデルでは各サーバは各々アドレス空間が独立したプロセスであることによる。したがって、サーバはあるジョブを実行する場合にメッセージベースのプロセス間通信機構を使用しなければならない。さらに、サーバプロセスとマイクロカーネル間のメッセージパッシングは、コンテキスト切り替えを要求するため、性能上のオーバヘッドを生じることになる。しかし、モノリシックカーネルモデルでは全てのサービスはカーネルによって提供され、同じアドレス空間で共有される。したがって、メッセージパッシングやコンテキスト切り替えはカーネルがジョブを実行する上で発生しない。そのため、モノリシックカーネルモデルはマイクロカーネルモデルより

も高速に処理できる。

この性能上の問題点があるにもかかわらず、マイクロカーネルモデルは最近の分散オペレーティングシステムの設計に好まれて採用されている。これに対する2つの主な理由を以下に示す。

1. マイクロカーネルモデルの利点が性能上の問題より優れている。このことは、高位プログラミング言語がアセンブリ言語より好まれていることに類似している。アセンブリ言語で書かれたプログラムの方が性能が良いのにもかかわらず、多くのプログラムは設計、保守及び移植性の観点から高位プログラミング言語によって書かれている。同様に、柔軟性に富むマイクロカーネルモデルはその性能の問題を上回っている。

2. 幾つかの実験的な結果から理論的には、マイクロカーネルモデルはモノリシックカーネルモデルより性能的に劣るが実際にはそうとは言えない。これは、他の要素の影響が大きいためメッセージ交換によるオーバヘッドは小さく、一般に無視できる程度となる [Douglis et al. 1991]。

マイクロカーネルモデルに基づく設計による分散オペレーティングシステムの詳細については第12章で述べる。

1.6.4 性能

分散システムが使用される場合、その性能 (**performance**) は少なくとも集中型システムと同じくらい良いものでなければならない。すなわち、特定のアプリケーションが分散システムで実行する場合、その全体の性能は単一プロセッサシステムで同じアプリケーションが実行するときよりも良いか又は同等くらいである必要がある。この目標を達成するために分散システムのオペレーティングシステムの各種の構成要素が適切に設計され、集中型システムよりも分散システムの性能が悪くならないようにすることが重要である。良い性能を達成するために有効な設計の原則は次のとおりである。

1. バッチの多用。バッチは、一般に性能の改善をもたらす。例えば、個々のページよりもむしろ大きくまとめた単位でネットワーク上をデータ転送した方がより効果的である。同様に、2つの通信エンティティがメッセージを交信する場合次のメッセージと一緒に前のメッセージのピギーバック**確認通知** (**piggybacking**) を行うことは性能を改善する。

2. キャッシュの活用。クライアントでのデータのキャッシュは、現在使用しているデータの可用性を高め、計算時間やネットワークの帯域を軽減するためにシステム全体の性能を向上させる。加えて、キャッシュは集中する資源への競合を軽減する。

3. データコピーの最少化。データコピーのオーバヘッド (例えばバッファとのデータの入出力) は、多くの動作を伴うために、CPU コストを高めてしまう。例えば、送信側から受信側にデータメッセージを転送する場合、送信側で次のような処理を伴う。

(a) 送信側の格納場所からメッセージバッファに移動

(b) 送信側のアドレス空間にあるメッセージバッファからカーネルのアドレス空間にあるメッセージバッファに移動

(c) 最後に、カーネルのアドレス空間にあるメッセージバッファからネットワークインタフェースボードに移動

受信側では、恐らくデータは逆の処理を行うであろう。したがって、この場合メッセージ転送動作において全体で 6 回のコピーが発生する。同様に、I/O 装置でブロックの読み出しや書き込み動作を行うとデータコピーのオーバヘッドが大きい。したがって、バッファの性能のために必ずしも簡単に達成できないがデータコピーを避けることが望まれる。メモリ管理機能を適切に使用することによってカーネル、ブロック I/O 装置、クライアント及びサーバ間のデータ移動を減少することができる。

4. ネットワークトラヒックの最小化。ノード間の通信を抑えることによってシステム性能を向上することができる。例えば、遠隔資源にアクセスする場合には恐らく中継ノードを介した通信が行われる。したがって、頻繁に利用する資源へのアクセスコストの低下が余り利用しない資源へのアクセスコストの増加に相殺できるならば、頻繁にアクセスする資源の近くにプロセスを移行することによってシステムのネットワークトラヒックを減少することが可能になる。ネットワークトラヒックを減少する別の方法として、システムの同じノード内で相互に通信を行うように複数のプロセスをまとめるためのプロセス移行機能を利用することである。ある決定を行う場合に全体の状態情報を集めることをしないことがネットワークトラヒックを抑えることができる。

5. 多重処理における細かい粒度の並行性の利用。性能は、多重処理における細かい粒度の並行性の利用によっても改善することができる。例えば、スレッ

ド (第 8 章で記述) は構造化サーバプロセスに対して良く用いられる。**スレッド (threads)** のグループとして構成されたサーバは、複数のクライアントからの要求を同時にサービスできるため効率良く動作できる。共有資源に対する複数のプロセスによる同時アクセスの細かい粒度の並列性制御は、よりよい性能に対する原則の別の応用事例である。

本書では、各種の分散オペレーティングシステム構成要素の設計においてこれらの設計原則に照らして考察する。

1.6.5 規模拡張性

規模拡張性 (scalability) は、サービスの負荷の増大に対応適用するためのシステムの能力である。企業において仕事の増加や組織的な変更に対応するために新たなマシンやサブネットワークをシステムに増強することは一般的であり、分散システムも刻々と成長することは避けられない。したがって、分散オペレーティングシステムはシステムのノードやユーザの増化に容易に対応できるように設計される必要がある。すなわち、このような増化に伴ってユーザへのサービスや性能の低下を招いてはならない。規模拡張性を持つ分散システムを設計するための指標となる原則は、次のものである。

1. **集中型エンティティの回避**:分散オペレーティングシステムの設計において、システム全体に対して単一の中央ファイルサーバやデータサーバなどの集中型エンティティの使用は、次のような理由から分散システムの規模拡張性に適用できない。

(a) 集中型エンティティの障害はシステム全体の障害をしばしば招く。そのため、システムは障害への対応がよくできない。

(b) ユーザの数が増加し、競合が多発すると集中型エンティティの性能はシステムのボトルネックになってしまう。

(c) 集中型エンティティが十分な処理能力と記憶容量を持っているとしても、集中型エンティティとシステムの他のノードが接続しているネットワーク能力は、エンティティへの競合がある一定以上になると飽和状態に陥る。

(d) 幾つかのローカルエリアネットワークを相互に接続する広域ネットワークにおいて幾つかのゲートウェイを介したサーバノードでサービスされるような特殊な要求を絶えず扱うことは、明らかに非効率である。これは、またネッ

トワークトラヒックの増加を招く。ローカルエリアネットワークと広域ネットワークの概念については第 2 章で述べる。

したがって、設計において集中型エンティティの使用は避けるべきである。これを達成するために資源の複製や分散制御アルゴリズムの技術がしばしば使用される。事実、規模拡張性に対応するために、システムの全てのノードができるだけ均等に役割を演じることができる機能的に対称となる構成法が用いられる。

2. 集中型アルゴリズムの回避:集中型アルゴリズムは、全ノードから情報を収集し、1 箇所で処理し、他のノードに結果を配布する機能を持つ。分散オペレーティングシステムの設計でこのようなアルゴリズムを使用することは、規模拡張性の観点からも受け入れ難い。この理由は、集中型エンティティの使用で述べたことと非常に似ている。例えば、全てのノードにまず問い合わせし、ジョブを引き受けられる候補として最も負荷が軽いノードを選択することによってスケジューリングの決定を行うスケジューリングアルゴリズムは、規模拡張性の弱点になる要素を持つことになる。このようなアルゴリズムは小規模なネットワークではうまく動作するが、大規模なネットワークでは弊害がでる。これは、問い合わせに対する応答が同時に非常に多くなるため処理するホストを選択するための時間が長くなりすぎてしまう。さらに、アルゴリズムの複雑度が $O(n^2)$ となるため、ネットワークトラヒックとして大きくなり、ネットワークの帯域が消費される。したがって、分散オペレーティングシステムの設計では、非集中型アルゴリズムのみが使用されるべきである。このようなアルゴリズムでは、システム全体の状態情報を収集することもなく、ローカルな情報に基づいて決定され、またシステム全体のクロックは存在しない (全てのノードのクロックの同期をとらなくてよい)。

3. クライアントのワークステーションでの動作の実行:可能ならば、オペレーションはできるだけ限りサーバマシンよりもクライアントのワークステーションで実行すべきである。すなわち、サーバは複数のクライアントの共有資源であり、サーバの処理はクライアントの処理よりも貴重である。この原則は、システムの規模が拡張するにつれてシステム性能を順次低下させていくことができるため、共有資源への競合を減少することによってシステムの規模拡張性にをがい改善できる。この原則を実現する技術としてしばしばキャッシュが使用される。

本書では、各種の分散オペレーティングシステム構成要素の設計においてこ

れらの設計原則に照らして考察する。

1.6.6 不均質性

不均質な分散システムは、異なったハードウェアやソフトウェアの相互接続の集合体として構成される。その多様性のために不均質な分散システムの設計は、各システムが同じ若しくは相互に関連したハードウェアやソフトウェアから構成される均質な分散システムの設計よりもはるかに困難である。しかしながら、大規模なシステムを考えた場合、不均質性は分散システムにおいて避けられないことである。さらに、不均質な分散システムは各種のアプリケーションに対する様様コンピュータプラットフォームのユーザに柔軟性を提供するため、**不均質性 (heterogeneity)** は多くのユーザの支持を得ている。例えば、シミュレーションでのスーパコンピュータ、文書処理での Macintosh、またプログラム開発での UNIX などの柔軟性をユーザは要求する。

不均質な分散システムにおける非互換性には様様なタイプのものがある。例えば、異なった通信やホストプロセッサでは内部的な形式が異なっている。また、ゲートウェイを介したネットワークでの相互接続では、ネットワークの通信プロトコルや構成が異なっている。また、システムの各々のノードでは動作するサーバが異なっている。すなわち、あるホストは 32 ビットのワード長を用い、他のものは 16 又は 64 ビットのワード長を用いる。これらのデータ構成のバイトオーダが異なるため、相互接続性がないホスト間でデータを共有するためには特別な変換が必要である。

不均質な分散システムにおいて、相互接続性のない 2 つのノード間で相互作用を持つためには何らかのデータ変換の形式が必要である。初期のシステムではこの機能をユーザに任せていたが、もはやこれは受け難いことである。このデータ変換の処理は、送信側のノードか受信側のノードで行われる。この処理が受信側のノードで行われるとすると、全てのノードで受信側で使用する形式にシステムで使用する各種の形式を変換しなければならない。したがって、n 個の異なる形式がある場合、各ノードでは $n-1$ 種の変換ソフトウェアを持つことになり、システム全体では $n(n-1)$ 種の変換ソフトウェアが必要になる。これは、新しい形式が加わると大変となるため非現実的である。受信側のノードではなく送信側のノードで行う場合でも同じような問題になる。

この変換処理のソフトウェアの複雑さは、中間での標準データ形式を用いることで大幅に減少できる。この方法では、中間での標準データ形式を定義し、

各ノードはそのノード自身のデータ形式から標準データ形式への変換とその逆の処理を行えば良いことになる。この場合、2 つの相互接続性のないノードが送信側のノードで相互作用を持つ時、送信するデータがまず標準の形式に変換され、標準形式で転送され、最後に受信側で標準形式から受信側の形式に変換される。システムに共通の標準形式を導入することで変換の数を減少することができる。

分散システムでの不均質性に関する各種の技術について第 2 章、第 4 章、第 5 章及び第 8 章で述べる。

1.6.7　セキュリティ

ユーザがシステムを信頼して期待できるようにするために、コンピュータシステムの各種の資源は破壊や不法なアクセスから守られなければならない。分散システムでのセキュリティを考える場合、制御が集中的でないことや安全が確保されてないデータ通信ネットワークを利用するため、集中型システムに比べて困難性が高まる。集中型システムでは、全てのユーザはログイン時にシステムによって認証され、ユーザがアクセスする資源を利用することが許されるか否か容易に審査できる。しかしながら、分散システムではクライアントサーバモデルが使用されるためクライアントがサーバに要求メッセージを送信したときにサーバはどのようなクライアントかを何らかの方法で調べなければならない。これは、クライアントの識別情報がメッセージに必ずあるとは限らないために、そう簡単にはいかない。これは、侵入者 (システムの資源に不正なアクセスを行う人又はプログラム) 正当なユーザになりすますことや転送するメッセージの内容を改竄することが考えられるができる。したがって、集中型システムと比べて、分散システムでの**セキュリティ (security)** の確保は次のようなことがさらに必要となる。

1. 送信者にとって送信するメッセージが目的とする受信者に受信されることの確認
2. 受信者にとって受信するメッセージが本物の送信者からのものであることの確認
3. 送信者、受信者の両者にとってメッセージの内容が転送中に改竄されていないことの確認

暗号 **(cryptography)**(第 11 章で述べる) は、分散システムでのセキュリティに対する最も実践的な方法である。この方法では情報を暗号化し、正しいユーザのみが復号化できることで、個人情報のセキュリティが確保される。

セキュリティに対する別の指標となる原則は、可能な限り少ないエンティティの完全性に基づいてセキュリティを確保することである。例えば、何千ものクライアントを保証するよりは小数のサーバの完全性に基づいてセキュリティを確保する方がより簡単になる。この場合、これらのサーバや実行するソフトウェアの物理的なセキュリティを保証することで十分である。第 11 章では安全な分散システムを設計するために共通的に使用される技術について述べる。

1.6.8 既存オペレーティングシステムのエミュレーション

市場で成功するためには、新しく分散オペレーティングシステムを設計する場合、既存のオペレーティングシステム、例えば UNIX のエミュレーションを行うことが重要である。この機能により、分散機能を十分に活用するために新しいオペレーティングシステムのシステム呼び出しインタフェースを用いて書くことができ、また既存の多くのソフトウェアを書き改めることなく同じシステムで実行することができる。したがって、新たな分散オペレーティングシステムへの移行によりこの両者のソフトウェアがそれぞれ動作可能となる。

第 12 章では既存の分散オペレーティングシステムがどのように UNIX のエミュレーション機能を実現しているかについて説明する。

1.7 分散計算環境 (DCE) の概要

本書の第 12 章では、4 つの分散オペレーティングシステム、Amoeba、V-System、Mach、Chorus について事例を説明する。さらに、業界標準になるような個々の分散オペレーティングシステム構成要素の鍵となる技術について各章で述べる。特に、次の技術については事例を説明する。

■第 2 章ではネットワーク技術の事例として、Ethernet、IEEE Token Ring、Internet protocol、インターネットを説明する。

■第 3 章ではメッセージパッシング技術の事例として、4.3BSD UNIX プロセス間通信機構を説明する。

■第 4 章では遠隔手続き呼び出し (RPC) 技術の事例として、SUN RPC と

DCE RPC について説明する。

■第 5 章では分散共有メモリ (DSM) 技術の事例として、IVY と Munin について説明する。

■第 6 章ではクロック同期技術の事例として、DCE 分散時間サービス (DTS) を説明する。

■第 8 章ではスレッド技術の事例として、DCE スレッドパッケージを説明する。

■第 9 章では分散ファイルシステム技術の事例として、DCE 分散ファイルサービス (DFS) を説明する。

■第 10 章では名前付け技術の事例として、DCE 名前付け機能の各種の構成要素について説明する。

■第 11 章ではセキュリティの技術の事例として、Kerberos 認証システムと DCE セキュリティサービスを説明する。

上記に示すように、本書の各章で述べる事例に含まれる鍵となる技術のほとんど半分は DCE に属するツールやサービスである。これは DCE がどのように創られたか (次の節で記述する) に起因している。したがって、これらの主要な技術を十分に理解するためには技術の詳細に入る前に DCE の概要を知ることが有効である。本章では **DCE(distributed computing environment)** の概要について述べる。

1.7.1　DCE とは

ベンダとは独立の**分散計算環境 (distributed computing environment)**、DCE は、IBM、DEC 及び Hewlett-Packard を含むコンピュータメーカのコンソーシアムである Open Software Foundation(OSF) によって定義された。これはオペレーティングシステムでもなければアプリケーションでもない。むしろ、既存のオペレーティングシステム上に共存する環境として実装され、分散アプリケーションを構築し実行するためのプラットフォームとして機能するサービスとツールの統合されたものである。

DCE の主要な目的は、ベンダから独立した仕様である。各種のベンダによって提供されるコンピュータ、オペレーティングシステム及びネットワークで実行できることである。例えば、DCE が容易に実装できるオペレーティングシステムとして、OSF/1、AIX、DOMAIN OS、ULTRIX、HP-UX、SINIX、

DCE アプリケーション
DCE ソフトウェア
オペレーティングシステム とネットワーキング

図 1.7 DCE による分散システムの DCE ソフトウェアの位置付け

SunOS、UNIX System、System V、VMS、WINDOWS、OS/2 が考えられる。一方、他の同様な製品と同様に TCP/IP、X.25 を含むネットワークハードウェアや通信ソフトウェアが利用できる。

図 1.7 に示すように、DCE は DCE アプリケーションとネットワーク層を含むオペレーティングシステム間に置かれるミドルウェアのソフトウェアである。基本的な考え方は、既存のマシン (恐らく各種ベンダ) の集合体であり、通信ネットワークによって相互に接続し、マシン本来のオペレーティングシステム上に DCE のソフトウェアプラットフォームを加え、分散アプリケーションを構築しかつ実行することを可能とすることである。各マシンは、他のマシンとは異なるオペレーティングシステムとしてそれ自身のローカルなオペレーティングシステムを持つ。ネットワーク層を含むオペレーティングシステム上の DCE ソフトウェア層は、必要に応じて自動的にデータタイプの変換を行うことでマシン間の差異を隠蔽する。したがって、システムの不均質性はアプリケーションプログラマに対して透過的であり、分散アプリケーションを書くことを簡単にする。

1.7.2 DCE はどのように開発されたか

OSF は零から DCE を開発したわけではない。言い換えれば、分散コンピューティングの分野において既に行われていた大学や業界の成果から DCE を創った。このために、OSF は技術要請 (RFT) を出し、首尾一貫した分散計算環境を構築するために必要となるツールやサービスを募った。競争するために、最終的には実際の動作すべきコードを提供することが基本的な要求であった。提案は、OSF のメンバや外部の専門家によって細心に評価された。最終的に、最良の解を与えると考えられるツールやサービスが評価委員会によって選定された。ツールやサービスに対するコードはほとんど C 言語で記述され、さら

に DCE として世界的に普及させるために単一の統合パッケージとして OSF によって開発された。DCE の 1.0 版が OSF から 1992 年 1 月に提供された。

1.7.3　DCE の構成要素

上述したように、DCE は個別に開発された各種の技術を組み合わせており、OSF によって巧く統合されている。これらの技術は DCE の構成要素を形成している。DCE の主要な構成要素は次のものである。

1. スレッドパッケージ (**threads package**):これは、並列アプリケーションを構築するための簡単なプログラミングモデルを提供する。この機能は、単一プロセス内での複数スレッドの生成と実行制御及びアプリケーション内のグローバルデータへのアクセスの同期制御のためのオペレーションを含む。詳細は第 8 章で述べる。
2. 遠隔手続き呼び出し (**Remote Procedure Call**)(**RPC**)):機能これは、クライアントサーバアプリケーションを構築するために必要となる数多くの強力なツールをプログラマに提供する。実際、DCE RPC 機能は、全ての DCE に基づくプログラミングモデルがクライアントサーバモデルであるために、DCE における通信の基礎となっている。これは、容易に利用でき、ネットワークやプロトコルに依存しない、クライアントとサーバ間の安全な通信を提供し、また自動的にクライアントとサーバに要求される適切な形式にデータを変換することでデータに対する相違を隠蔽する。詳細は第 4 章で述べる。
3. 分散時間サービス (**Distributed Time Service**)(DTS):これは、システム内の全てのコンピュータのクロック同期をとる。また、これは外部の時間とシステム内のコンピュータのクロック同期をとるために、例えば米国の国立標準技術研究所 (NIST) の時計を利用するなど外部の時計の値を利用できることを可能にする。この機能によって、異なる分散環境のコンピュータ同士の時刻を合わせることができる。詳細は第 6 章で記述する。
4. 名前サービス (**Name Service**):DCE の名前サービスは、セルディレクトリサービス (CDS)、グローバルディレクトリサービス (GDS) 及びグローバルディレクトリエージェント (GDE) を含む。これらのサービスは、サーバ、ファイル、装置などの資源に対して位置に依存しない方法でユニークな名前付けやアクセスを可能にする。詳細は第 10 章で述べる。
5. セキュリティサービス (**Security Service**):これは、不正なアクセスから

構成要素名	使用される他の構成要素
スレッド	なし
RPC	スレッド、名前、セキュリティ
DTS	スレッド、RPC、名前、セキュリティ
名前	スレッド、RPC、DTS、セキュリティ
セキュリティ	スレッド、RPC、DTS、名前
DFS	スレッド、RPC、DTS、名前、セキュリティ

図 **1.8** DCE 構成要素の従属関係

システム資源を守るために認証や承認に必要となるツールを提供する。詳細は第 11 章で述べる。

6. **分散ファイルサービス (Distributed File Service)**(DFS):これは、位置に依存しない、高性能な、高可用性を持つシステムワイドのファイルシステムを提供する。DCE DFS の特徴ある機能は、他のファイルシステムのクライアントに対してもファイルサービスを提供する。詳細は第 9 章で述べる。

上記に示した DCE の構成要素は、密に関係している。相互に関係しているため個別に絵で描くことは困難である。例えば、名前サービスはそのサービスの中で内部的に RPC 機能を利用しており、また RPC 機能も宛先の位置特定のために名前サービスを利用している。したがって、各種の DCE 構成要素の相互関係は図 1.8 のように表形式によって表現できる。

1.7.4 DCE セル

DCE システムは、DCE を実行するシステムが世界中の数千のコンピュータや数百万のユーザを対象にすることができる点で大規模なものである。このような大規模なシステムを可能にするために DCE はセル **(cell)** の概念を持つ。この概念は、大規模なシステムをセルと呼ぶ小さくて管理し易い単位に分割する。

DCE システムでは、セルは共通的な目的を持ち、共通の DCE サービスを共有するユーザ、マシン若しくはその他の資源のグループである。最小のセルの構成は、セルディレクトリサーバ、セキュリティサーバ、分散時間サーバ及び 1 つ以上のクライアントマシンを含む。各 DCE クライアントマシンは、セキュ

リティサービス、セルディレクトリサービス、分散時間サービス、RPC 機能及びスレッド機能のためのクライアントプロセスを持つ。また、DCE クライアントマシンはセルに DCE 分散ファイルサーバがあるならば分散ファイルサービスのためのプロセスを持つ。クロック同期 (clock synchronization)(第 6 章で述べる) 重複法に通知する方式を使用するためには、DCE システムの各セルは少なくとも 3 つの分散時間サーバを持たなければならない。

DCE システムの立ち上げ時の最も重要なことは、セルの境界を決定することである。この決定を行うためには次の 4 つの項目を考慮することが必要である [Tanenbaum 1995, Rosenberry et al. 1992, OSF 1992]。

1. 目的:共通の目標に向かって作業しているユーザのマシンは、同じセルに置かれるべきであり、これにより共通のシステム資源へのアクセスが容易になる。すなわち、同じセルのマシンのユーザは他のセルのユーザとはより密な相互動作を持つ。例えば、会社で各種の製品を製造し、販売する場合、その会社に応じて製品指向か又は機能指向かの方法によってセルの境界が決定される [Tanenbaum 1995]。製品指向の方法では、各製品に応じてセルが分離され、1 つのセルに所属するマシンのユーザは 1 つの特定の製品に関する全ての作業 (設計、製造、販売、保守) に責任を持つことになる。一方、機能指向の方法では、各セルは各作業に応じて分離され、そのセルに所属するユーザは全ての製品の特定の作業、例えば設計に対して責任を持つ。

2. 管理:各システムでは、ユーザの加入やそのユーザのシステム資源へのアクセス権を決定するための管理者が必要となる。管理者は、この作業を適切に行うためにシステムのユーザ及び資源を知らなければならない。管理者の仕事をわかり易くするためには、管理者が認識し、管理し易くするために対象となるマシンやユーザを同じセルに置くことが必要である。例えば、企業又は大学の同じ部に所属するマシンは同じセルに所属する。管理者の観点から見ると、各セルはそれぞれ別の管理者によって管理される。

3. セキュリティ:お互いに信頼関係のあるユーザのマシンは、同じセルに置くべきである。すなわち、セル内のマシンのユーザ同士は、他のセルのユーザよりは信頼関係が強い。例えば設計において、セル内の資源にアクセスするよりも他の資源にアクセスする場合の方がチェックが厳しくなるため、セルの境界は一種の防火壁の役割を果たす。

4. オーバヘッド:幾つかの DCE オペレーション、例えば名前の解決やユーザ認証は、同じセルで処理されるよりも異なるセル間で処理される方がオーバ

ヘッドは大きくなる。したがって、頻繁に相互作用を持つユーザのマシンや資源は同じセルに置くべきである。他のセルの資源へのアクセスは、システム全体の性能を考慮して少なくするようにすべきである。

上記では目的、管理、セキュリティ、オーバヘッドの観点からセルの境界を論じているが、セルの設計において地理的条件も考慮することが必要である。性能の観点からすると、セル境界をまたがるオペレーションの数を最小にするためにはできるだけセルの数を少なくすることが望まれる。しかしながら、セキュリティや管理の面からは少ないマシンやユーザを持つ小さなセルが望まれる。したがって、DCE システムにおけるセルの境界を決める上では上述した 4 つの項目をバランス良く考慮することが重要である。

1.8 まとめ

分散コンピューティングシステムは、通信ネットワークによって相互に接続されたプロセッサの集合であり、各プロセッサはそれぞれローカルメモリや周辺装置を持ち、他のプロセッサと通信ネットワークを介してメッセージを交信するための通信機能を備えている。

分散コンピューティングシステムに対する既存のモデルは、ミニコンピュータ、ワークステーションサーバ、プロセッサプール、及びハイブリッドの大きく 5 つに分類できる。 分散コンピューティングシステムは、従来の集中型システムよりも複雑であり、構築が困難である。このように複雑で構築が困難にもかかわらず、分散コンピューティングシステムは急速に導入され、利用されている。これは分散コンピューティングシステムの利点がその欠点を凌いでいることによる。分散コンピューティングシステムの主な特長は、(a) 本来の分散アプリケーションに対する適応性、(b) 分散したユーザ内での情報の共有、(c) 資源の共有、(d) 高い性能価格比、(e) 短い応答時間と高いスループット、(f) 高信頼性、(g) 規模拡張性と段階的な規模適応性、及び (h) ユーザ要望に対する柔軟性である。

分散コンピューティングシステムに共通的に使用されるオペレーティングシステムは、ネットワークオペレーティングシステムと分散オペレーティングシステムの 2 つに大きく分類される。ネットワークオペレーティングシステムに比べて、分散オペレーティングシステムはより良い透過性とフォールトトレラント能力を持ち、ユーザに仮想的な単一プロセッサの機能を提供する。

分散オペレーティングシステムの設計における主な課題は、透過性、信頼性、柔軟性、性能、規模拡張性、不均質性、セキュリティ及び既存オペレーティングシステムのエミュレーションである。

1.9 演習問題

1.1 下記の 4 つのオペレーティングシステムのタイプの違いをそれぞれの特長を定義して述べよ。

(a) タイムシェアリング

(b) 並列処理

(c) ネットワーク

(d) 分散

1.2 分散コンピューティングシステムが並列処理システムより優れている点は何か？ 分散コンピューティングシステムが並列処理システムより適している 3 種類のアプリケーションの例を挙げよ。

1.3 分散コンピューティングシステムが開発された動機として、主な技術的、経済的、社会的要素は何か？ 分散コンピューティングシステムが集中型と比べて持つ主な利点と欠点は何か？

1.4 分散コンピューティングシステムを構成するために共通的に使用される幾つかのモデルの利点と欠点を述べよ。将来どのようなモデルが最も普及すると思うか？ またその理由を述べよ。

1.5 プールとして P 個のプロセッサを持つプロセッサプールモデルに基づく分散コンピューティングシステムの場合を考える。このシステムで、ユーザは F 個のソースファイル ($F ¡ P$) からなるプログラムの計算を開始する。この時そのユーザはシステムで一人の利用者であるとする。このシステムでこの処理を行う場合単一プロセッサシステムに比べて最大処理速度としてどの程度期待できるか (全てのプロセッサは同じ性能とする)？ この最大処理速度より低下する要因は何か？

1.6 サービスとサーバの用語の違いを説明せよ。分散オペレーティングシステムの設計において、サービスを実現する上で単一のサーバと複数のサーバを使用する利点と欠点を述べよ。

1.7 分散オペレーティングシステムを設計する場合、集中的なタイムシェアリングシステムのオペレーティングシステムの設計に比べて何故難しいか？

1.8 グループウェアとは何か？ これはソフトウェア開発において何故有望な

技術と考えられているか？

1.9 ネットワークオペレーティングシステムと分散オペレーティングシステムの主な相違は何か？

1.10 分散オペレーティングシステムを設計する上での主な課題は何か？

1.11 分散オペレーティングシステムはネットワークで接続されるマシンを1つの仮想単一プロセッサとして動作させる。この仮想マシンアーキテクチャはユーザに対して何が利点か？ この目的を達成する上で分散オペレーティングシステム設計者にとって重要な課題は何か？

1.12 並列透過性は分散オペレーティングシステムを設計する上で重要な課題である。これは、集中型システムのオペレーティングシステムを設計する場合にも重要な課題か？ もしそうであれば、この機能を実現するために集中型システムのオペレーティングシステムで共通的に使用される機構を列挙せよ。

1.13 分散オペレーティングシステム設計者がシステムの信頼性を向上するための重要な概念を述べよ。システムを高信頼にする上での主な問題は何か？

1.14 分散オペレーティングシステムを設計する上でのモノリシックカーネルとマイクロカーネルの方法の相違について、利点と欠点の関係を述べよ。

1.15 分散オペレーティングシステムの設計をマイクロカーネルで実現する場合、カーネルが行わなければならない主な機能は何か？

1.16 図1.6は階層化の方法が分散システムの設計に使用されることを示しているが、この方法による利点は何か？

1.17 分散オペレーティングシステム設計者がシステムの性能を確保するための主な指標となる原則について述べよ。

1.18 分散システムの設計で何故規模拡張性は重要な機能であるか？ 拡張可能な分散システムを設計する上での指標となる原則について述べよ。

1.19 何故多くの分散システムでは不均質性を避けられないか？ 不均質な分散システムにおいて共通的な非互換性は何か？

1.20 突然分散システムがクラッシュした場合、次のような状況ではユーザにどのような不具合を与えるか？

(a) システムがプロセッサプールモデルで、クラッシュしたものがプールのプロセッサである。

(b) システムがプロセッサプールモデルで、クラッシュしたものがユーザの端末である。

(c) システムがワークステーションサーバモデルで、クラッシュしたものがサーバマシンである。

(d) システムがワークステーションサーバモデルで、クラッシュしたものがユーザのワークステーションである。

1.21 次のシステムについて、価格、ハードウェアの複雑性、オペレーティングシステムの複雑性、並行性及びプログラミングのし易さ (ユーザがどのように容易で効率的にプログラムが作成できるか) の観点で比較せよ。

(a) 単一の共有メモリを持つマルチプロセッサシステム

(b) システム全体で使用できる共有メモリに加えて各プロセッサがローカルのメモリを持つマルチプロセッサシステム

(c) 各プロセッサは各々メモリを持つマルチプロセッサシステム。全てのプロセッサはまとめられ、高帯域な通信回線によって接続されネットワークを形成する。各プロセッサは他のプロセッサとメッセージの交信によってのみ通信ができる。

(d) 各プロセッサは各々メモリを持つマルチプロセッサシステム。プロセッサはお互いに遠く離れ (別の都市など)、低速な通信回線によって接続されネットワークを形成する。各プロセッサは他のプロセッサとメッセージの交信によってのみ通信ができる。システムを比較する上で、次の 3 つの場合を考える。(a) プロセッサの数が小さい (2~8)、(b) プロセッサの数が大きい (16~32)、(c) プロセッサの数がとても大きい (100 以上)

参考文献

[Accetta et al. 1986] Accetta, M., Baron, R., Golub, D., Rashid, R., Tevanian, A., and Young, M., "Mach: A New Kernel Foundation for UNIX Development," In: *Proceedings of the Summer 1986 USENIX Technical Conference*, pp.93-112 (July 1986).

[Avresky and Pradhan 1996] Avresky, D., and Pradhan, D. (Eds.), *Fault-Tolerant Parallel and Distributed Systems*, IEEE Computer Society Press, Los Alamitos, CA (1996).

[Black et al. 1992] Black, D. L., Golub, D. B., Julin, D. P., Rashid, R. F., Draves, R. P., Dean, R. W., Forin, A., Barrera, J., Tokuda, H., Malan, G., and Bohman, D., "Microkernel Operating system Architecture and Mach," In: *Proceedings of the USENIX Workshop on Microkernels and Other Kernel Architectures*, USENIX, pp. 11-30 (1992).

[Boykin and LoVerso 1990] Boykin, j., and LoVerso, J., "Recent Developments in Operating Systems," *IEEE Computer*, pp. 5-6 (May 1990).

[**Brazier and Johansen 1993**] Brazier, F., and Johansen, D. (Eds.), *Distributed Open Systems*, IEEE Computer Society Press, Los Alamitos, CA (1993).

[**Butler 1993**] Butler, M., *Client Server*, Prentice-Hall, London, UK (1993).

[**Casavant and Singhal 1994**] Casavant, T. L., and Singhal, M. (Eds.), *Readings in Distributed Computing Systems*, IEEE Computer Society Press, Los Alamitos, CA (1994).

[**Cheriton 1984**] Cheriton, D. R., "The V Kernel: A Software Base for Distributed Systems," *IEEE Software*, Vol. 1, No. 2, pp. 19-42 (1984).

[**Cheriton 1988**] Cheriton, D. R., "The V Distributed System," *Communications of the ACM*, Vol. 31, No. 3, pp. 314-333 (1988).

[**Coulouris et al. 1994**] Coulouris, G. F., Dollimore, J., and Kindberg, T., *Distributed Systems Concepts and Design*, 2nd ed., Addison-Wesley, Reading, MA (1994).

[**Cristian-1991**] Cristian, F., "Understanding Fault-Tolerant Distributed Systems," *Communications of the ACM*, Vol. 34, pp. 56-78 (February 1991).

[**Critchley and Batty 1993**] Critchley, T., and Batty, K., *Open Systems: The Reality*, Prentice-Hall, London, UK (1993).

[**Deitel 1990**] Deitel, H. M., *An Introduction to Operating Systems*, 2nd ed., Addison-Wesley, Reading, MA (1990).

[**Douglis et al. 1991**] Douglis, F., Ousterhout, J. K., Kaashoek, M. F., and Tanenbaum, A. S., "A Comparison of Two Distributed Systems: Amoeba and Sprite," *Computing Systems*, Vol. 4, pp. 353-384 (1991).

[**Ghafoor and Yang 1993**] ghafoor, A., and Yang, J., "A Distributed Heterogeneous Super-computing Mnagement System," *IEEE Computer*, Vol. 26, No. 6, pp. 78-86 (1993).

[**Gien 1990**] Gien, M., "Micro-kernel Architecture: Key to Modem Operating Systems Design," *UNIX Review*, p. 10 (November 1990).

[**Gien and Grob 1992**] Gien, M., and Grob, L., "Microkernel Based Operating Systems: Moving UNIX on to Modern System Architectures," In: *Proceedings of the UniForum'92 Conference*, USENIX, pp. 43-55 (1992).

[Golub et al. 1990] Golub, D., Dean, R., Forin, A., and Rashid, R., "UNIX as an Application Program," In:*Proceedings of the Summer 1990 USENIX Conference*, USENIX, pp. 87-95 (June 1990).

[Goscinski 1991] Goscinski, A., *Distributed Operating Systems, The Logical Design*, Addison-Wesley, Reading, MA (1991).

[Hariri et al. 1992] Hariri, S., Choudhary, A., and Sarikaya, B, "Architectural Support for Designing Fault-Tolerant Open Distributed Systems," *IEEE Computer*, Vol. 25, No. 6, pp. 50-62 (1992).

[Hunter 1995] Hunter, P., *Network Operating Systems: Making the Right Choice*, Addison-Wesley, Reading, MA (1995).

[Islam 1996] Islam, N., *Distributed Objects: Methodologies for Customizing Operating Systems*, IEEE Computer Society Press, Los Alamitos, CA (1996).

[ISO 1992] *Basic Reference Model of Open Distributed Processing, Part 1: Overview and Guide to Use*, ISO/IEC JTC1/SC212/WG7 CD10746-1, International Standards Oragnization (1992).

[Jalote 1994] Jalote, P., *Fault Tolerance in Distributed Systems*, Prentice-Hall, Englewood Cliffs, NJ (1994).

[Khanna 1994] Khanna, R. (Ed.), *Distributed Computing: Implementation and Management Strategies*, Prentice-Hall, Englewood Cliffs, NJ (1994).

[Khokhar et al. 1993] Khokhar, A., Prasanna, V. K., Shaaban, M. E., and Wang, C. L., "Heterogeneous Computing: challenges and Opportunities," *IEEE Computer*, Vol. 26, No. 6, pp. 18-27 (1993).

[Lampson 1983] Lampson, B. W., "Hints for Computer System Design," In: *Proceedings of the 9th Symposium on Operating Systems Principles* (October 1983).

[Lamport et al. 1982] Lamport, L., Shostak, R., and Pease, M., "The Byzantine Generals Problem," *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 3, pp. 382-401 (1982).

[Lelann 1981] Lelann, G., "Motivations, Objectives, and Characterization of Distributed Systems," *Distributed Systems—Architecture and Implementation, Lecture Notes in Computer Science*, Vol. 105, Springer-Verlag, New York, NY (1981).

[Lockhart Jr. 1994] Lockhart Jr., H. W., *OSF DCE: Guide to Deveroping Distributed Applications*, IEEE Computer Society Press, Los Alamitos, CA (1994).

[Marca and Bock 1992] Marca, D., and Bock, G. (Eds.), *Groupware: Software for Computer-Suported Cooperative Work*, IEEE Computer Society Press, Los Alamitos, CA (1992).

[Martin et al. 1991] Martin, B. E., Pedersen, C. H., and roberts, J. B., "An Object-Based Taxonomy for Distributed Computing Systems," *IEEE Computer*, Vol. 24 No. 8 (1991).

[Milenkovic 1992] Milenkovic, M., *Operating Systems: Concepts and Design*, 2nd ed., McGraw-Hill, New York (1992).

[Mullender 1987] Mullender, S. J.,"Distributed Operating Systems," *Computer Standards and Interfaces*, Vol. 6, pp.37-44 (1987).

[Mullender 1993] Mullender, S. J. (Ed.), *Distributed Systems*, 2nd ed., Addison-Wesley, Reading, MA (1993).

[Mullender and Tanenbaum 1984] Mullender, S. J., and Tanenbaum, A. S., "Protection and Resource Control in Distributed Operating Systems," *Computer Networks*, Vol. 8, pp.421-432 (1984).

[Mullender et al. 1990] Mullender, S. J., Van Rossum, G., Tanenbaum, A. S., Van Renesse, R., and Van Staverene, H., "Amoeba: A Distributed Operating System for the 1990s." *IEEE Computer*, Vol. 23, No. 5, pp. 44-53 (1990).

[Needham and Herbert 1982] Needham, R. M., and Herbert, A. J., *The Cambridge Distrigbuted Computing System*, Addison-Wesley, Reading, MA (1982).

[Nelson 1990] Nelson, V. P., "Fault-Tolerant Computing: Fundamental Concepts," *IEEE Computer*, Vol. 23, No. 7, pp. 19-25 (1990).

[Nicol et al. 1993] Nicol, J. R., Wilkes, C.T., and Manola, F.A., "Object Orientation in Heterogeneous Distributed Computing Systems," *IEEE Computer*, Vol. 26, No. 6, pp.57-67 (1993).

[Notkin et al. 1987] Notkin, D., Hutchinson, N., Sanislo, J., and Schwartz, M., "Heterogeneous Computing Environments: Report on the ACM SIGOPS Workshop on Accommodating Heterogeneity." *Communications of the ACM*, Vol. 30, No. 2, pp. 132-140 (1987).

[Nutt 1991] Nutt, G. J., *Centralized and Distributed Operating Systems*, Prentice-Hall, Englewood Cliffs, NJ (1991).

[Nutt 1992] Nutt, G. J., *Open Systems*, Prentice-Hall Englewood Cliffs, NJ (1992).

[OSF 1992] *Introduction to OSF DCE*, Prentice-Hall, Englewood Cliffs, NJ (1992).

[Ousterhout et al. 1988] Ousterhout, J. K., Cherenson, A. R., Douglis, F., Nelson, M. N., and Welch, B. B., "The Sprite Network Operating System." *IEEE Computer*, Vol. 21, No. 2, pp.23-36 (1988).

[Pike et al. 1990] Pike, R., Presotto, D., Thompson, K., and Trickey, H., "Plan 9 from Bell Labs," In: *Proceedings of the Summer 1990 UKUUG(UK Unix Users Group) Conference*, pp. 1-9 (July 1990).

[Popek and Walker 1985] Popek, G., and Walker, B., *The LOCUS Distributed System Architecture*, MIT Press, Cambridge, MA (1985).

[Rashid 1985] Rashid, R. F., "Network Operating Systems," In: *Local Area Networks: An Advanced Course, Lecture Notes In Computer Science*, Vol. 184, pp. 314-340, Springer-Verlag, New York, NY (1985).

[Ritchie and Thompson 1974] Ritchie, D., and Thompson, K., "The UNIX Time-Sharing System," *Communications of the ACM*, Vol. 17, No. 7, pp. 365-375 (1974).

[Rosenberry et al. 1992] Rosenberry, W., Kenney, D., and Fisher, G., *OSF DISTRIBUTED COMPUTING ENVIRONMENT, Understanding DCE*, O'Reilly, Sebastopol, CA (1992).

[Schlichting and Schneider 1983] Schlichting, R. D., and Schineider, F. B., "Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems," *ACM Transactions on Computer Systems*, Vol. 1, No. 3, pp. 222-238 (1983).

[Shoch and Hupp 1982] Shoch, J. F., and Hupp, J. A., "The Worm Programs: Early Experiences with a Distributed Computation," *Communications of the ACM*, Vol.25, No. 3, pp. 172-180 (1982).

[Silberschatz and Galvin 1994] Silberschatz, a., and Galvin, P. B., *Operating systems Concepts*, 4th ed., Addison-Wesley, Reading, MA (1994).

[Singhal and Shivaratri 1994] Singhal, M., and Shivaratri, N. G., *Advanced Concepts in Operating Systems*, McGraw-Hill, New York, NY (1994).

[Stalling 1995] Stalling, W., *Operating Systems*, 2nd ed., Prentice-Hall, Englewood Cliffs, NJ (1995).

[Stankovic 1984] Stankovic, J. A., " A Perspective on Distributed Computer Systems," *IEEE Transactions on Computers*, Vol. C-33, No.12, pp. 1102-1115 (1984).

[Suri et al. 1994] Suri, N., Walter, C. J., and Hugue, M. M. (Eds.), *Advances in Ultra-Dependable Distributed Systems*, IEEE Computer Society Press, Los Alamitos, CA (1994).

[Tanenbaum 1995] tanenbaum, A. S., *Distributed Operating Systems*, Prentice-Hall, Englewood Cliffs, NJ (1995).

[Tanenbaum and Van Renesse 1985] Tanenbaum, A. S., and Van Renesse, R., "Distributed Operating Systems," *ACM Computing Surveys*, Vol. 17, No. 4, pp. 419-470 (1985). ©ACM, Inc., 1985.

[Umar 1993] Umar, A., *Distributed Computing: A Practical Approach*, Prentice-Hall, Englewood Cliffs, NJ (1993).

[Vaughn 1994] Vaughn, L. T., *Client/Seraver System Design and Implementation*, IEEE Computer Society Press, Los Alamitos, CA (1994).

[Wittie 1991] Wittie, L. D., "Computer Networks and Distributed Systems," *IEEE Computer*, pp. 67-75 (1991).

インターネット参考文献

オペレーティングシステムに関する参考文献は、以下のところで見つけることができる。
ftp:ftp.cs.umanitoba.ca/pub/bibliographies/Os/os.html

並列分散システムの分類法に関する参考文献は、以下のところで見つけることができる。
ftp:ftp.cs.umanitoba.ca/pub/bibliographies/Parallel/taxonomy.html

分散コンピューティングに関する参考文献は、以下のところで見つけることができる。
ftp:ftp.cs.umanitoba.ca/pub/bibliographies/Distributed/Osser.html

分散システムに関する参考文献は、以下のところで見つけることができる。
ftp:ftp.cs.umanitoba.ca/pub/bibliographies/Distributed/Dcs-1.0.html

ftp:ftp.cs.umanitoba.ca/pub/bibliographies/Distributed/dist.sys.1.html

オープンシステムとオープンコンピューティングに関する参考文献は、以下のところで見つけることができる。
ftp:ftp.cs.umanitoba.ca/pub/bibliographies/Os/opencomp.html

フォールトトレラント分散システムに関する参考文献は、以下のところで見つけることができる。
ftp:ftp.cs.umanitoba.ca/pub/bibliographies/Distributed/fault.tolerant.html

コンピュータ支援共同作業 (CSCW) に関する参考文献は、以下のところで見つけることができる。
ftp:ftp.cs.umanitoba.ca/pub/bibliographies/Distributed/CSCWBiblio.html
ftp:ftp.cs.umanitoba.ca/pub/bibliographies/Distributed/CSCW92/html

MIT の並列分散オペレーティングシステム (PDOS) の刊行リストは、以下のところで見つけることができる。
http://www.pdos.lcs.mit.edu/PDOS-papers.html

Stanford の分散システムグループ (DSG) の刊行リストは、以下のところで見つけることができる。
http://www-dsg.stanford.edu/Publications.html

Oregon GraduateInstitute 分散システム研究グループ (DSRG) の刊行リストは、以下のところで見つけることができる。
http://www.cse.ogi.edu/DSRG/osrg/osrg.html/Current Paper

ダブリンの Trinity 大学の分散システムグループの刊行リストは、以下のところで見つけることができる。
http://www.dsg.cs.tcd.ie/dsgpublications/bibs

第2章

コンピュータネットワーク

2.1 概要

コンピュータネットワークは、ネットワークのエンドシステム上で動作する2つのプロセス間のデータの交信を行うために、通信回線やプロトコルソフトウェアによってエンドシステムを接続する通信システムである。エンドシステムは、ノード **(node)** 、サイト **(site)** 、ホスト **(host)**、コンピュータ **(computer)**、マシン **(machine)** などと呼ばれる。ノードは大きさや機能によって様々である。大きさの面で見ると、ノードには小さなマイクロプロセッサ、ワークステーション、ミニコンピュータ又は大規模なスーパコンピュータがある。機能面では、ユーザとの相互作用を持たない専用のシステム (例えば、プリンタサーバやファイルサーバ)、ユーザ用のパーソナルコンピュータ又は汎用のタイムシェアリングシステムがある。

既に第1章で述べたように、分散システムは基本的にローカルメモリを持ち、また他のハードウェアやソフトウェアの資源をもつノードから構成されるコンピュータネットワークである。したがって、分散システムは構成するノード間のデータや制御情報の通信のためにコンピュータネットワーク上に実現される。さらに、分散システムの性能や信頼性は下位のコンピュータネットワークの性能や信頼性に依存する。従って、分散オペレーティングシステムを学習するためにはコンピュータネットワークの基本知識が必要となる。コンピュータネットワークをしっかりと理解するためには、専門書が必要となるが、多くの良書が出されている [Tanenbaum 1988, Black 1993, Stallings 1992b, Ramos et al. 1996]。 従って、本章では分散オペレーティングシステムの設計の基礎と

して必要となる事柄にしぼってネットワークを構成するための概念と設計について述べる。

2.2 ネットワークのタイプ

ネットワークは、大きく2つのタイプ、ローカルエリアネットワーク **(local area networks)(LAN)** と広域ネットワーク **(wide area networks)(WAN)** に分類される。また、WAN は長距離ネットワーク **(long-haul networks)** とも呼ばれる。これらの2つのタイプのネットワークの特徴的な相違は次のものである [Abeysundara and Kamal 1991]。

1. **地理的分散 (geographic distribution)**: 2つのネットワークタイプの主な相違の1つは、地理的な広がりにある。LAN は数キロメートルの限られた範囲を対象とし、WAN は数千キロメートルを越える広範囲なネットワークである。したがって、LAN は建物やキャンパス内の通信機能を提供し、WAN は国内若しくは世界規模のネットワークを対象とする。

2. **データ速度 (data rate)**: データ伝送速度は、通常 WAN よりも LAN の方が高速である。LAN の伝送速度は、通常 0.2 メガビット／秒 (Mbps) から1ギガビット／秒 (Gbps) である。一方、WAN の伝送速度は通常 1200 ビット／秒から 1 Mbps 以上までくらいである。

3. **誤り率 (error rate)**: ローカルエリアネットワークは、一般的に WAN よりもデータ伝送誤りは少ない。一般には、LAN のビット誤り率は 10^{-8}〜10^{-12} で WAN は 10^{-5}〜10^{-7} である。

4. **通信リンク (communication link)** : LAN で使用される代表的な通信媒体は、ツイストペア、同軸ケーブル及び光ファイバである。一方、WAN では広域のため使用される通信媒体は比較的低速で信頼性が低い。WAN での代表的な通信媒体は、電話回線、マイクロウェーブ回線及び衛星回線である。

5. **所有関係 (ownership)**: LAN は限られた地域をカバーするため、一般に単一の組織に所属する。しかしながら、WAN は一般に複数の LAN から構成され、それぞれの LAN は異なる組織に所属する。したがって、通常 WAN よ

りも LAN の方が管理や保守が複雑でなく、費用がかからない。

6. **通信費用 (communication cost)**: LAN の全体の通信費用は、通常 WAN よりも少ない。これは、誤り率が小さく、ルーティングアルゴリズムが単純 (若しくはない)、さらに管理や保守の費用が少ないことにやる。さらに、LAN でのデータ通信費用はユーザの設備であるためほとんど無視できる。しかしながら、WAN では使用する伝送媒体は専用回線若しくは公衆通信システム、例えば電話回線、マイクロウェーブ回線、衛星回線であるため、かなり高い。

LAN と WAN の両者の特徴の一部を兼ね備えたネットワークは、**メトロポリタンエリアネットワーク (metropolitan area networks)(MAN)** と呼ばれる [Stallings 1993a]。MAN は、通常 LAN よりも大きな範囲 (直径約 50 km まで) をカバーし、LAN と同程度の速度を提供する。MAN の主な目的は、街全体若しくは大都市の LAN を相互接続することである。MAN で一般に使用される通信リンクは、同軸ケーブルやマイクロウェーブ回線である。

第 1 章で既に述べたように、分散システムの性能は少なくとも集中型システムと同等でなければならない。すなわち、特定のアプリケーションを分散システム上で実行した場合、同じアプリケーションを単一プロセッサシステムで実行した場合よりも性能が悪化することは許されない。LAN や MAN のデータ伝送速度は、通常多くのアプリケーションに対してこの要求を満足していると考えられる。しかしながら、現在の技術では WAN のデータ伝送速度は十分に分散システムの要求を満たすことができない。したがって、WAN を利用した分散システムは、主に性能が重要視されないアプリケーションに適用される。広範囲なユーザやコンピュータ間で情報の共有を行う分散アプリケーションの一部はこの分野に属する。

これまでの WAN は分散システムの性能の要求を十分に満足するものではないが、広帯域統合サービスデジタルネットワーク (B-ISDN)[Kawarasaki and Jabbari 1991] や非同期転送モード (ATM) 技術 [Vetter 1995] の出現によって、将来の WAN は分散システムの幅広いアプリケーションに対応できるデータ伝送速度を提供することが期待できる。ISDN[Helgert 1991] は、各種のデータ、例えば音声、ファックスやコンピュータのデータを基本チャネル速度 64 キロビット／秒 (Kbps) の倍数の速度でディタル伝送できる電気通信ネットワークである。B-ISDN は 150 Mbps 以上のデータ伝送速度を提供する ISDN である。B-ISDN は、広帯域な通信を必要とするアプリケーション、例えば高品質なビ

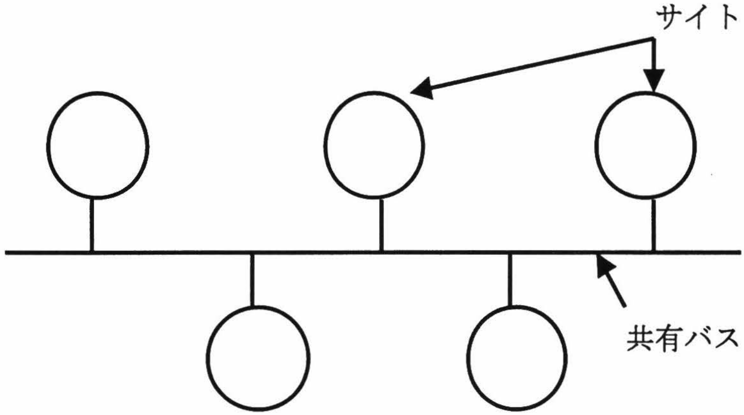


図 2.1　単一多重アクセスバスネットワーク形態

デオやバルクデータ伝送などのアプリケーションに適用されると考えられる。ATM 技術は、622 Mbps までのデータ伝送速度を提供できる (ATM 技術については本章の後半で述べる)。

2.3　LAN 技術

この節は、代表的な LAN の構成、動作原理及び事例について述べる。

2.3.1　LAN の構成

LAN を構成するためのネットワーク構成として、多重アクセスできるバスとリングの 2 つがある。単純な**多重アクセスバスネットワーク (multiaccess bus network)** では、全てのサイトは直接ネットワーク全体の長さまで伸びる単一の伝送媒体 (バスと呼ぶ) に接続される (図 2.1)。バスは受動系であり、ネットワークでのメッセージ伝送において全てのサイトによって共有される。各サイトは T 型コネクション又はタップを使った分岐ケーブルによってバスに接続される。メッセージ伝送のためにブロードキャスト通信が用いられる。すなわち、あるサイトから他のサイトへのメッセージは共有のバス上に伝送される。宛先のアドレスがメッセージに付与させる。メッセージがバス上を伝送され、各サイトは自分宛かをチェックし、そうであればメッセージを受信する。

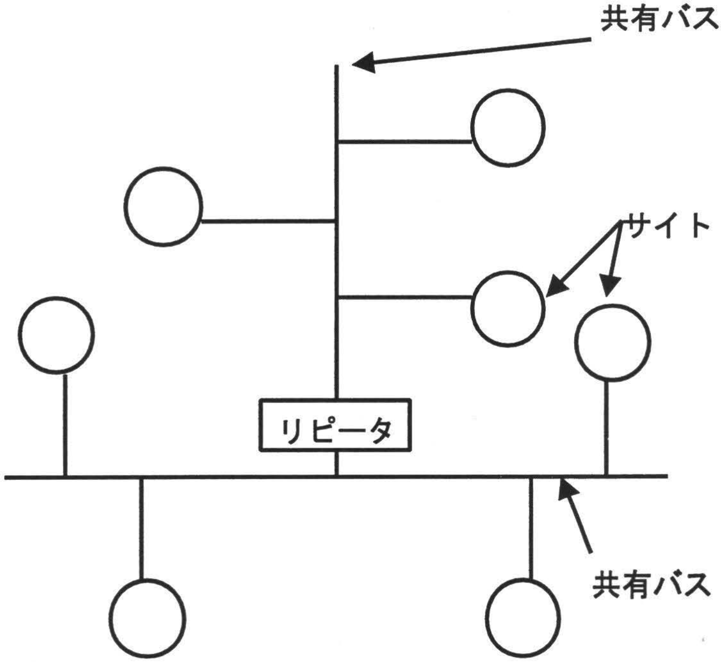


図 2.2 多重アクセス分岐バスネットワーク形態

単純な多重アクセスバスネットワーク構成の拡張として、多重アクセス分岐バスネットワーク構成がある。このネットワークは、複数の単純な多重アクセスバスネットワークがリピータによって接続されたものである (図 2.2)。**リピータ (repeater)** は、ケーブルのセグメントを接続するためのハードウェア装置である。これは、単に一方のセグメントからの電気信号を増幅し、他方のセグメントに電気信号を伝える。

多重アクセスバスネットワークの接続費用は小さく、サイトの数の増加に従って大きくなる。通信費用も、共有バス上での競合が大きくなって、バスがボトルネックとならない限り小さい。多重アクセスバス構成の欠点は、単一の共有メディア上で伝送されるメッセージの信号が他の 2 地点間リンクの構成に比べて歪みの影響を受け易いことである。したがって、特別な場合として光ファイバによるバスネットワークをサポートするサイトがある。

リングネットワーク (ring network) では、各サイトは両方のサイトと接続され、ループを形成する (図 2.3)。2 つのサイト間に個々のリンクが使用される。

リンクはリピータによって接続される。データは、各サイトを信号として一方向に伝送される。すなわち、あるサイトから他のサイトへのメッセージの送信は、送信元のサイトがメッセージヘッダの中に宛先のサイトアドレスを書き込み、隣のサイトに渡すことによって行われる。メッセージを受信したサイトは、自分宛のメッセージかを調べるためにメッセージヘッダをチェックする。自分宛でなければ、メッセージを隣のサイトに渡す。この方法によって、メッセージはリングから取り除かれるまでリング上を巡回する。メッセージを取り除く方法として、宛先サイト(メッセージの宛先)がリングから取り除く場合と、送信元サイト(メッセージの送信元)がリングから取り除く方法がある。後者の場合では、メッセージは必ずリングを一周する。一般に、サイトの中の1つは**監視サイト (monitor site)** としてメッセージが無限に(すなわち、送信元サイトや宛先サイトの障害によって)巡回しないように監視する。この監視サイトは、他の機能として、構成管理機能、リング状態管理機能、障害管理機能など持つ。

リングネットワークの接続費用は小さく、サイトの数の増加に比例して大きくなるにしぎない。平均的な通信費用は、サイトの数に比例して増加する。n サイトある場合、メッセージは宛先に届くために最大 $(n-1)$ リンクを伝送する。しばしば引用されるリング構成の欠点は、サイト又はリンク障害がリングネットワーク全体に影響を与えることである。基本リング構成の拡張として、2つのサイト間を二重リンクで接続し、両方向の伝送リンクとする方法があり、障害箇所をループバックしてネットワークの信頼性を向上するものである。

2.3.2　メディアアクセス制御プロトコル

多重アクセスバスやリングネットワークの場合、単一のチャネルがネットワーク上の全てのサイトによって共有され、多重アクセス環境となることを既に述べた。このような環境において、幾つかのサイトが同時に共有チャネル上で情報を伝送することが考えられる。この場合、伝送された情報は壊され、廃棄されなければならない。情報が廃棄されたことを伝送したサイトに通知すると、サイトは再送できる。もし特別な対策がなければ、この状況は繰り返され、性能が低下してしまう。したがって、共有チャネルをアクセス制御するための特別な方式が多重アクセス環境においては必要になる。このような方式は、**メディアアクセス制御プロトコル (medium-access control protocol)** と呼ばれる。明らかに、多重アクセス環境では、伝送路のデータ伝送速度が速いことだけでは十分ではない。メディアアクセス制御プロトコルは、伝送路の効果的

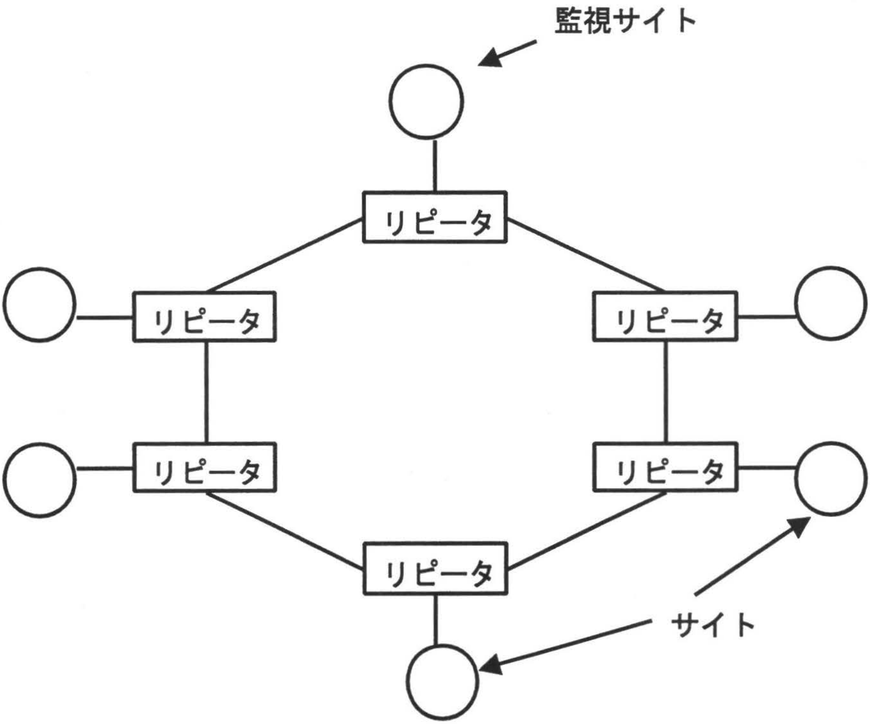


図 **2.3** リングネットワーク形態

な帯域使用を実現することが重要である。したがって、メディアアクセス制御プロトコルはコンピュータネットワーク全体の性能に重大な影響を与え、このプロトコルによってネットワークも変わってくる。メディアアクセス制御プロトコルの性能に関する3つの重要な目的は、高スループット、高チャネル使用率及び低メッセージ遅延である。この性能の目的に合致することに加えて、メディアアクセス制御プロトコルに望まれる他の特徴として下記のものがある[Abeysundara and Kamal 1991]。

1. 公平性に対して、優先方式が意識的に導入されない限り、プロトコルとしては共有の伝送路上で情報を伝送する機会を全てのサイトに公平に与えるべきである。

2. 規模拡張性に対して、サイトがネットワーク構成 (構成、大きさ又は他の

サイトとの位置関係) を意識することは最小限度にとどめ、サイトの追加、削除及び移動はプロトコルの変更なしに行えるべきである。さらに、プロトコルが正しく機能する上でエンドツーエンドの伝搬遅延を意識しなくても良いことが必要である。

3. 信頼性に対して、集中制御は避けるべきであり、プロトコル動作は完全に分散すべきである。

4. リアルタイムアプリケーションに対して、プロトコルは遅延特性を明確にすべきである。すなわち、ネットワーク内のあるサイトから他のサイトへの最大メッセージ伝送遅延は明らかにし、固定とすべきである。

高スループット、高チャネル使用率及び低メッセージ遅延の性能の目的をかなえつつ、同時に上記の特徴を全て実現することは困難である。

幾つかのプロトコルが、多重アクセス環境におけるメディアアクセス制御のために開発されている。その中で、衝突検出機構付き搬送波センス多重アクセス (CSMA/CD) プロトコルが多重アクセスバスネットワークで最も普及しており、リングネットワークではトークンリングとスロッテッドリングの 2 つがある。これらについては以下で述べる。

ネットワークの資源を有効にかつ公平に使用するため、メッセージはパケットに分割されて送られる。この場合、パケット **(packet)** が伝送上の最小の単位である。これは、ヘッダと本体を持つ。本体には実際のメッセージのデータを、ヘッダにはアドレス情報 (送信と受信のサイトを示す) とシーケンス番号 (1 つのメッセージ内のパケットの順番) を含む。

CSMA/CD プロトコル

CSMA/CD (Carrier Sense Multiple Access with Collision Detection) 方式 [IEEE 1985a] は、共有伝送路での分散制御を採用している。この方式では、各サイトは集中制御サイトが存在しないため対等である。サイトはお互いに共有メディアの使用権を競争し、メディアが使用されていなければ最初にアクセス権を得たサイトがメッセージを伝送する。この場合、複数のサイトで伝送路が使用されていないと判断し、同時にメッセージを送信するとメッセージの衝突が発生する。この方式では、この衝突を検出し、この問題の回復のために再送機構を持っている。したがって、この方式は以下に述べる 3 つの機構を持つ。

1. 搬送波センス (**carrier sense**) と延期機構 (**defer mechanism**): あるサイトがパケットを伝送する場合、まず共有伝送路上の信号 (無線による放送との類似性から搬送波と呼ばれる) の有無を調べる。伝送路が空き状態 (搬送波 (**carrier**) が存在しない) であれば、サイトはパケットの伝送を開始する。そうでなければ、サイトはパケットの伝送を延期し、メディアが空き状態になるまで待つ (状態の監視は続ける)。伝送路の状態が空きになればすぐにパケットの伝送を開始する。

2. 衝突検出機構 (**collision detection mechanism**): しかしながら、共有伝送路の伝搬遅延が 0 でないため、搬送波センスは全ての衝突を回避できない。確かに、衝突は伝送を開始した後の短期間のみに発生し、この期間後は全てのサイトは伝送路の状態が空きでないことを感知し、伝送を延期する。この期間は衝突ウィンドウ (**collision window**) 又は衝突期間 (**collision interval**) と呼ばれ、信号が共有伝送路の端から端を往復に伝搬する時間に等しい。サイトはパケットを伝送した場合、そのパケットが衝突を起こしていないか確認するため衝突期間と等しい時間共有伝送路の状態を調べなければならない。

パケットを伝送する前に少なくとも衝突期間の間共有伝送路の状態を調べることが必要となるため、衝突が希であっても伝送路の使用率は低下してしまう。したがって、衝突を避けることに代えて、CSMA/CD 方式は衝突の発生を許容し、それを検出して必要な回復機構を備えている。

分散衝突検出機構として、サイトは出力ポートからパケットを伝送すると入力ポートの信号を読み、出力と入力の 2 つの信号を比較する方法を採用している。衝突を検出するとパケットの伝送を停止し、他のサイトに衝突が発生したことを通知するためにジャミング信号 (**jamming signal**) と呼ばれる特別の信号を共有伝送路の送出する。このジャミング信号を受信すると全てのサイトはその時に受信していたパケットを廃棄する。衝突したパケットを伝送したサイトは、ある時間経過後そのパケットを再送する。

全ての衝突の検出を確実にするために、あるパケット長が必要となる。これを説明するために 2 つのサイト間の最大伝搬遅延を t とする。2 つのサイトが同時にパケットを伝送すると、少なくとも他のサイトのパケットを受信し、衝突を検出するためには時間 t 後となる。そのため、パケット長が短いとそのパケットを全て送出するに必要な時間は時間 t より小さくなり、パケットを受信する前に 2 つのサイトはパケットの伝送を完了するため衝突の検出ができなくなる。しかしながら、同じネットワークにいる他のサイトは、2 つのサイトか

らの伝搬時間が t より小さいため、2つのサイトが伝送したパケットの衝突を受信する。

3. 制御された再送機構 (**controlled retransmission mechanism**):衝突した後、壊れたパケットは再送されなければならない。衝突のためにパケットが壊れた場合、ジャミング信号を受けてすぐにパケットの再送を全てのサイトが行うと恐らく再度衝突が発生する。衝突の再発を最小にしかつ過負荷状態でのチャネルを安定にするために、制御された再送方式が使われている。この方式では、共有伝送路での競合を分散アルゴリズムを使って解決している。

再送の方針として、相反する2つの目標、(a) パケットをすばやく伝送しかつ共有伝送路の使用効率を高めるための再送のスケジューリングと (b) 混雑しているメディアでのサイトの負荷を軽減するための自律的なバッキングオフがある。再送アルゴリズムとして、パケットを再送する前に遅延を計算する方法がとられる。衝突が発生すると、少なくとも衝突期間よりも長い遅延時間を再送する前に各サイトでとることである。この時間を**再送スロット時間 (retransmission slot time)** と呼ぶ。伝送路の使用を早めるためにこのスロット時間はできるだけ短い方が良いが、衝突を避けるために衝突期間よりも長くなければならない。したがって、スロット時間は通常伝送路のラウンドトリップ時間よりも少し長く設定される。リアルタイム遅延時間は、再送遅延D(正の整数) と再送スロット時間 (S_t) の積である。

制御された再送機構の良い事例は、イーサネットで使用されているバイナリ指数バックオフアルゴリズムである。このアルゴリズムは、イーサネットの説明 (LAN 技術の事例) としてこの節で述べる。

CSMA/CD 方式は、バーストでかつ非同期的な転送を行うバス構成のネットワークに適している。実現性やチャネル使用率の点から比較的低速 (10 Mbps 程度) な伝送路に向いている。CSMA/CD 方式の性能は、伝搬遅延に対するパケット長の比率に依存する。伝搬遅延はパケットが衝突を起こす期間であるため、この比率が高くなれば性能は良くなる。この期間を過ぎると、全てのサイトは伝送するか待っていることになる。この比率が低下すると衝突の頻度が高まり、著しく性能の低下を招く。伝搬遅延に対するパケット長の比率が小さ過ぎると性能が決定的に悪くなるため、CSMA/CD 方式は高速度若しくは長距離の LAN には適していない。すなわち、光の伝送路 (数百メガビット／秒又はギガビット／秒) に対しては CSMA/CD 方式は受け入れ難い。

高速度、短パケット及び長いケーブルのシステムに不適切であることに加え

て、CSMA/CD は次のような欠点を持っている。(a) 遅延が明確に決まらない。共有伝送路の負荷によって遅延が変化するため、メッセージを伝送する時にネットワークの負荷が高ければ特定の時間内にメッセージを伝送する保証ことができない。(b) 共有伝送路上での優先機能が提供できない。全てのサイトが対等であるため、あるアプリケーションで特定のサイトの利用が高くても他のサイトよりも優先することができない。

トークンリングプロトコル

この方式は、共有伝送路における集中制御方式を採用している。この方式では、共有伝送路へのアクセスはシステム内のサイトを巡回している単一のトークンによって制御される。**トークン (token)** はメッセージの特別なタイプ (ユニークなビット列) で、これを保有するとメッセージを共有伝送路に伝送できる。トークンの特定のフィールドはフリーかビジーかを示す。トークンは、あるサイトから隣のサイトに一方向にリングを巡回する。メッセージを伝送できる準備ができているサイトは、トークンが届いてかつそれがフリーであるまで待たなければならない。フリートークンが到着すると、そのトークンをビジーにし、そのトークンにメッセージをつけてリング上の次のサイトに伝送する。受信するサイトは、トークンの状態をチェックする。トークンがフリーの場合にはそのサイトのメッセージを伝送できる。フリーでない場合は、ビジートークンに付いているメッセージの宛先が自分宛かを調べる。自分宛であればメッセージを受信し、メッセージを取り除いたトークンをリングの次のサイトに送る。リングを一周してビジートークンが送信したサイトに戻ってくると、リングからビジートークンを取り除き、新たにフリートークンを生成して次のサイトに転送し、メッセージの伝送を可能にする。フリートークンは、メッセージを伝送するサイトに届くまでリング上のサイトを巡回する。1 つのサイトが長時間トークンを保有することを防ぐために、トークン保持タイマがサイトがトークンを持つことができる時間を制御するために使用される。信頼性のある動作を保証するためにトークンの喪失や重複に対して対策がとられなければならない。すなわち、サイトの障害によってトークンが喪失した場合、システムはこれを検出し、新しいトークンを生成する。これは通常監視サイトが行う。さらに、サイト i が障害を起こした場合、サイト $i-1$ が直接サイト $i+1$ にトークンを送ることができるようにリングを再構成しなければならない。

トークンリングプロトコル (token ring protocol) の利点は、衝突が発生しないためメッセージの遅延が明確になることである。別の利点は、可変長の

パケットと同様長いパケットや短いパケットを伝送できる点である。原理的には、トークンに付加できるメッセージ長には制限がない。しかしながら、大きな欠点は負荷が軽い時でもフリートークンを受信するまでの初期待ち時間があることである。この初期待ち時間は、特に大きなリングでは無視できない。

上で述べたトークンリングプロトコルと別なものとしてIEEE802.5標準トークンリングプロトコルがある [IEEE 1985c]。このプロトコルについては、この節の後半でIEEEトークンリングの説明 (LAN技術の事例) で述べる。

スロッテッドリングプロトコル

この方式は、固定メッセージ長のスロットが固定数分連続にリング上を巡回している。各スロットは、制御とデータの2つの部分を持つ。制御部分は、通常スロットが空いているか否かを示すフィールド、スロットにメッセージがある場合にはそのメッセージの送信元と宛先のアドレス、さらにメッセージが宛先に正しく届いたかの情報が含まれる。一方、データ部分には固定長のメッセージを含むことができる。

メッセージを送信するサイトは、まずメッセージをスロットのデータの大きさにパケット化する。そして、空きのスロットの到着を待つ。空きのスロットが到着すると、それを獲得し、データ部分へのメッセージの一部であるパケットの設定、送信元及び宛先アドレスの設定、使用中／空きフィールドを使用中に設定し、リングに送出する。このスロットはリング上を巡回し、サイトは別の空きスロットの到着を待つ。サイトはメッセージの残りのパケットの伝送が完了するまでこの動作を行う。

各サイトは、全てのスロットについて自分宛のものであるかを調べる。自分宛でなければ、単にリング上の次のサイトにスロットを転送する。自分宛であれば、スロットからパケットを取り出し、宛先に正しく届いたことを示す情報を設定し、リング上の次のサイトにスロットを転送する。スロットがリングを一周して送信元に戻ってくると、スロットの使用中／空きフィールドを空きに変更し、他のメッセージの伝送を可能にする。

この方式は、リングの偏った使用を防ぎ、全てのサイトが共有伝送路を公平に使用することを保証する。また、この方式は同時に複数のサイトがメッセージを送信することを可能にする (トークンリング方式では、ある時点において1つのサイトのみが送信できる)。しかしながら、長いメッセージは小さなパケットに分割することが必要となる (このことはトークンリング方式では制約とはならない)。

このスロッテッドリング **(slotted-ring protocol)** 方式は、1970 年代にケンブリッジ大学で開発された Cambridge Ring[Wilkes and Wheeler 1979] に採用され、英国で広く使われている。このプロトコルの詳細については文献 [King and Mitrani 1987] で見ることができる。

2.3.3 LAN 技術の事例：イーサネットと IEEE トークンリング

イーサネット

イーサネット **(Ethernet)** は、比較的高速で経済的であるため分散システムを構築する上で LAN 技術として最も広く使われている。これは、DEC(Digital Equipment Corporation)、Intel および Xerox によって 1989 年に発表され、その後一部の修正を行い、IEEE 802.3[IEEE 1985a] 標準と呼ばれる標準 LAN 技術として IEEE に採択された。

イーサネットのネットワーク構成は、単純な多重アクセスバス若しくは多重アクセス分岐バスの構成である。通信媒体は、データ速度 10 Mbps の低損失な同軸ケーブルである。メッセージはパケット (イーサネットでは**フレーム (frame)** と呼ばれる) に分割され、バスにブロードキャストされることで、あるサイトから他のサイトに伝送される。バス上を伝送されるパケットを各サイトは自分宛かを調べ、そうであればパケットを読み込む。

多重アクセス制御として、イーサネットは制御された再送機構としてバイナリ指数バックオフアルゴリズム **(binary exponential back-off algorithm)** を持つ CSMA/CD プロトコルを使用する。イーサネットのバイナリ指数バックオフアルゴリズムにおいて、再送遅延 (D) の値は 0 から上限値 (L) の間にある特定の再送期間の値からランダムに決められる。すなわち、2 つのサイトが衝突を起こした場合、1 つのサイトは $X \times (S_t)$(S_t は再送スロット時間) 時間後に再送し、他方のサイトは $Y \times (S_t)$ 後に再送する。ここで、X と Y はランダムに選ばれるため異なる値となる。共有伝送路を制御し、高負荷でも安定に保つために、L の値は引く続いて衝突が起こると前の値の 2 倍になり、D の値の選択する範囲が広がる。イーサネットにおいて、最初の衝突では D の値は 0 か 1 のどちらかであり、2 番目の衝突では D の値は 0、1、2、3 から選ばれる。i 番目の衝突では D の値は 0 から $2^i - 1$ の範囲からランダムに選ばれる。送信が失敗 (衝突のくり返し) すれば、それだけ再送する前の時間間隔が長くなる可能性が大きくなる。このアルゴリズムは、最初は短い再送遅延で行い、しだいに大きくなるため伝送路が過負荷状態になることを防ぐ。

6バイト	6バイト	2バイト	46バイトから1500バイト	4バイト
宛先アドレス	送信元アドレス	タイプ	メッセージデータ	検査合計

図 **2.4**　イーサネットのパケット構造

何度か再送を繰り返すと再送期間は大きくなる。過度の遅延や遅い応答を避け伝送路特性を改善するために、再送を停止する前に再送期間を倍にすることは通常あるレベルで止める方法がとられ、何度かの再送を行う。この方式は**上限付バイナリ指数バックオフアルゴリズム (truncated binary exponential back-off algorithm)** と呼ばれる。イーサネットワークのパケット構成を図 2.4 に示す。宛先と送信元アドレスはそれぞれ 6 バイトである。宛先アドレスは、単一のサイトアドレス (単一のサイトを示す)、マルチキャストアドレス (サイトのグループを示す) 又はブロードキャストアドレス (全サイトを示す) である。宛先アドレスの各ビットが全て 1 のアドレスはブロードキャストアドレスである。あるマルチキャストアドレスに所属するサイトは、そのマルチキャストアドレス宛のパケットを受信できるようにネットワークインタフェースを構成する。さらに、単一サイトアドレスとマルチキャストアドレスを識別するためマルチキャストアドレスの最上位ビットは必ず 1 であり、単一サイトアドレスの最上位ビットは必ず 0 である。タイプフィールドは、パケットのタイプを識別するために使用される。このフィールドは、上位層の通信プロトコル (通信プロトコルの階層については本章の後半で述べる) によって使用される。

メッセージデータフィールドは、パケット構成の中で唯一可変長である。これは伝送される実際のメッセージデータを含む。このフィールド長は、46 バイトから 1500 バイトである。このフィールドが最小 46 バイト必要であるのは、メディアアクセス制御として CSMA/CD 方式による衝突の検出を保証するためである。一方、このフィールドが最大 1500 バイトになっているのは、ネットワークに関するバッファを大きくし、できるだけ送信サイトが一度に大きなパケットを送ることによって次の送信までの待ち時間を小さくするためである。メッセージデータフィールド長を示すフィールドはなく、パケットの終わりを検出するために 2 つのパケット間には 9.6 マイクロ秒の間隔が置かれる。

パケットの最後の 4 バイトは、送信側で生成された検査合計を必ず含み、受

信側で受信したパケットの正当性を検査する。受信側は、不正な検査合計のパケットを単に捨てる。このため、イーサネットではメッセージの送達は保証されない。このことがローカルネットワークで使用されている簡単なデータグラムプロトコルの信頼性が高くないことの理由である。メッセージの送達の保証が必要であれば通信プロトコル (通信プロトコルの階層については本章で述べる) の上位層で各パケットに対して応答を行い、無応答の場合にはタイムアウトによる再送を行わなければならない。

全てのイーサネットハードウェアインタフェースは、製造時にユニークなアドレスが付与される。これによってイーサネットで接続される全てのサイトは必ずユニークなアドレスを持つ。IEEE はイーサネットのアドレスの割り当てを管理している。48 ビットのあるビットの範囲がイーサネットハードウェアインタフェースの製造会社に割り当てられる。

IEEE トークンリング

分散システムを構築するために共通的に使用されるもう一つの LAN 技術として、IEEE トークンリング技術があり、IEEE 802.5 標準 [IEEE 1985b] と呼ばれる。IBM はこの技術を分散システムの製品に採用している。

IEEE トークンリング技術を使用するネットワークの構成はリング構成であり、そのメディアアクセス制御プロトコルはトークンリングプロトコル (**token ring protocol**) である。当初 4 Mbps の速度であったが、その後 16 Mbps まで拡張された。これは、伝送路として安価なツイストペア若しくは光ファイバを使用でき、全てのサイトが送信する場合にも帯域の無駄は発生しない。

3 バイトからなる 1 つのトークン (**token**) がリング上を巡回している。トークンはビジーかフリーである。メッセージを送信したいサイトは、フリーのトークンがそのサイトに到着するとメッセージをトークにつなぎ、そのトークンをビジーに変更する。したがって、ビジートークンはメッセージパケットを含む図 2.5 に示す構成である。

トークンの最初の 1 バイト目は、パケットの先頭を認識し、伝送速度の同期合わせのための特定のビット列である。2 バイト目は、優先度とトークン状態 (フリー／ビジー) 情報を持つ。3 バイト目は、パケット制御情報である。優先度フィールドはネットワークのサイトがチャネル容量を共有するための各種の方法を実現するために使用できる。

トークンに加えて、IEEE トークンリングのパケットは、送信元と宛先アドレス (各 6 バイト)、メッセージデータ (長さは 5000 バイト以下)、検査合計、

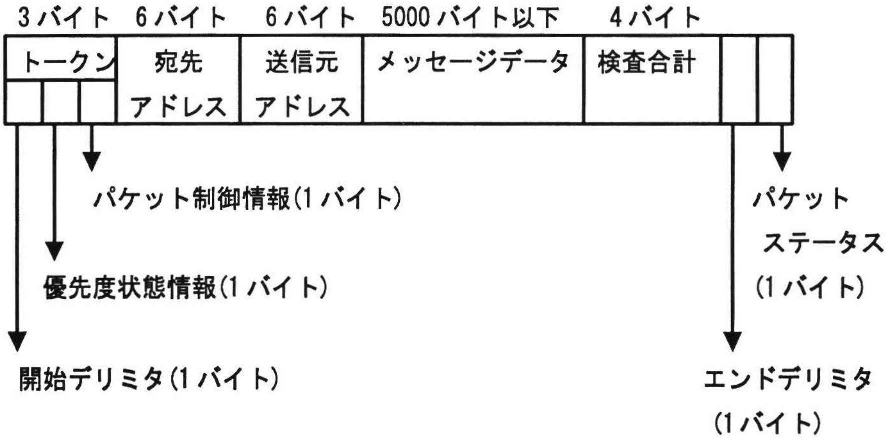


図 **2.5**　IEEE トークンリングのパケット構造

エンドデリミタ (1 バイト) 及びパケット状態のフィールドを持つ。送信元アドレスと宛先アドレスのフィールドは、それぞれ送信側のサイト及び受信側のサイトのアドレスを含む。メッセージデータフィールドは伝送するデータを含む。このフィールドは、可変長のパケットを含む。このフィールド長の上限は、5000 バイトが既定値であるが、初期設定により決めることができる。検査合計フィールドは、送信側が生成した検査合計を含み、受信側が受信したパケットの正当性を調べるために使用する。エンドデリミタフィールドは、パケットの終わりを識別するための特定のビット列を含む。最後のパケット状態フィールドは、パケットが受信側で正しく受信されたかを示す。このフィールドによって送信側は、パケットを受信側が受信したかを知ることができる。送信側のサイトは送信した後一周して戻ったパケットを取り除く責任がある。

2.4　WAN 技術

コンピュータの広域ネットワーク (**wide area networks**) は、遠く離れたコンピュータを相互に接続することによって構成される。これらのコンピュータは、いろいろな都市や国に置かれている。一般に、**WAN(wide area networks)** を利用するコンピュータの相互接続のための決まった構成はない。また、WAN の通信回線に使用される通信媒体はいろいろなものがある。例え

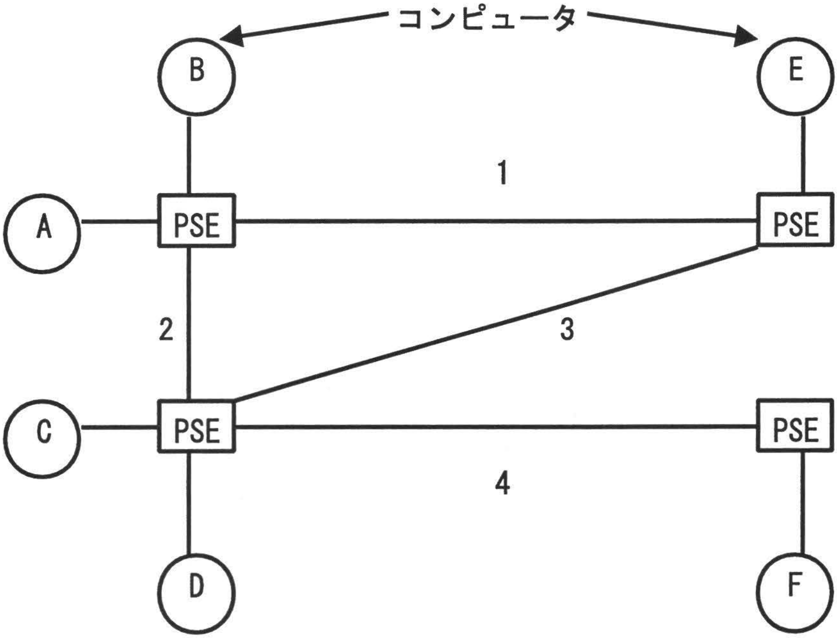


図 **2.6** コンピュータの WAN 構成

ば、WAN では同じ国のコンピュータは通信ケーブル (電話回線) で相互に接続され、国が異なるコンピュータは通信衛星によって相互に接続される。

WAN を利用するコンピュータは相互に直接通信チャネルで接続されないで、データ通信の目的のための専用のコンピュータであるパケット交換機 **(packet-switching exchanges) (PSE**) と呼ばれるハードウェア装置に接続される。したがって、ネットワークの通信チャネルは実際にネットワーク上のデータ通信機能を実行する PSE を接続する (図 2.6)。WAN でのコンピュータは、他のコンピュータとのメッセージの送受信のために接続されている WAN の PSE とのみやり取りを行う。

ネットワークにおいて、他のコンピュータにメッセージパケットを送信するために、コンピュータは接続されている PSE にパケットを送信する。パケットは、送信側のコンピュータの PSE から受信側のコンピュータの PSE に幾つかの他の PSE を経由して転送される。パケット転送の実際の方式や送信側のコンピュータの PSE から受信側のコンピュータの PSE に転送するための経路は、

ネットワークのPSEの交換技術やルーティング技術に依存している。これらの技術の各種の方法については次に説明する。パケットが受信側のコンピュータのPSEに届くと、パケットはその受信側のコンピュータに配送される。

2.4.1　交換技術

WANの通信は、送信元のコンピュータから宛先のコンピュータに複数のPSEを経由しパケットを転送することで行われる。PSEは、宛先にパケットを届くまで、あるPSEから別のPSEにパケットが転送されるための交換機能を提供する。すなわち、PSEは入力チャネルからのパケットを出力チャネルに転送する。ネットワークの遅延は、WANのPSEに使用される交換技術に強く依存する。最も共通的な2つの方式は、回線交換とパケット交換である。これらについて以下に述べる。

回線交換 (circuit switching)

この方式では、データ転送が開始される前に回線確立フェーズで送信側と受信側のコンピュータ間に物理的な回線が確立される。このフェーズにおいて、回線を確立するチャネルは専用的に設定され、ルーティングPSEにはバッファは必要とされない。回線が一度確立すると全てのデータのパケットは、データ転送フェーズで転送される。全てのメッセージデータのパケットはルーティングサイトでバッファリングされることなく転送されるため、パケットは連続するデータストリームとして扱われる。最後に、最後のパケットが転送されると回線切断フェーズで回線が解放される。回線が解放されると次の回線交換のために利用可能になる。要求した回線がビジー (使用中) のために回線が確立できない場合、回線が輻輳であると呼ばれる。輻輳の回線の扱い方に依存して、再度の回線確立のために一部の回線が解放されるかもしれない。

この方式は、公衆電話網に近い方式である。公衆電話網では、電話の発信により、発呼側の電話と着呼側の電話の間に電話交換局によって専用の電話回線が確立する。一度回線が確立すると、通信中の遅延は、ケーブルや交換機を介して伝送される電磁信号の伝搬遅延である。時として回線の確立ができない場合 (例えばクリスマスの日の長距離電話) があるが、一度回線がつながると電話を切るまでその通話は確保される。

回線交換技術の主な利点は、一度回線が確立するとほぼ無視できる程度の伝搬遅延以外には遅延が発生しない点である。さらに、回線の全容量が接続され

るコンピュータの対にそれぞれ占有的に使用されるため、メッセージを伝送するために必要となる時間が回線が確立された後は計算でき、かつ保証される。しかしながら、この方式は回線確立と回線切断のフェーズのオーバヘッドを持ち、接続されるコンピュータの対によって有効的に回線が使用されなければその回線容量は無駄になる。したがって、この方式は長い間連続的に伝送する場合や最大伝送遅延を保証する場合にのみ適用できると考えられる。この方式は、音声や分散アプリケーションにおけるリアルタイムデータの伝送に向いている。

回線交換は、公衆電話交換網 (PSTN) に使用されている。

パケット交換 (packet switching)

この方式では、送信側と受信側の (コンピュータの) 対に専用の通信パスを確立する代わりに、チャネルが別の送受信対のパケットの伝送のために共有される。すなわち、チャネルは送受信対のメッセージの 1 つのパケットを伝送する時にのみ占有され、チャネルは他の送受信対や同じ送受信対の別のパケットの伝送に使用される。

この方式では、各パケットは宛先のコンピュータのアドレスを含み、他のパケットとは独立に宛先に送ることができる。このため、同じメッセージ内のパケットでもネットワーク上の別の経路で送られることもあり、宛先のコンピュータでは送られたパケットの順番とは異なった順番で受け取ることもある。したがって、宛先側でメッセージの再組み立てを行う必要がある。パケットが PSE に届くと、パケットは一時的にパケットバッファに置かれる。次のチャネルが利用可能となり、また隣接の PSE のパケット受信バッファが用意できると、パケットは次の隣接の PSE に転送される。したがって、パケットの転送毎に経路が決まるため宛先までの実際のパケットが転送される経路は動的である。パケット交換技術は、**蓄積交換 (store-and-forward)** の通信として知られており、各パケットは次の PSE に転送される前に PSE に一時的に蓄積される。

回線交換に比べて、パケット交換はもともと小さなデータの塊の転送するのに適している。この方式は、1 つのチャネルの伝送帯域を複数のメッセージの転送のために共有するためチャネルの効率的な使用が可能となる。さらに、パケットによって実際の経路が動的に選択されるためネットワークの信頼性が高まり、障害となった PSE や経路を避けて別の経路を選択することができる。例えば、図 2.6 の WAN でチャネル 2 が障害になってもコンピュータ A と D は経路 1、3 を使って通信を継続することができる。しかしながら、PSE でのパケットバッファの追加や宛先コンピュータでのパケットの再組み立て等のオー

バヘッドが生じる。したがって、この方式は大量のデータ転送には有効ではない。この方式の欠点は、伝送時間がパケットの経路やデータ量に依存するため送信元のコンピュータから宛先のコンピュータまでのメッセージの伝送時間が保証できないことである。

2.4.2　ルーティング技術

WANにおいて、送信元と宛先のコンピュータの間に複数の経路が存在する場合、パケットを転送するために何れか1つが選ばれる。例えば、図2.6のWANにおいてコンピュータEとFの間には3–4と1–2–4の経路があり、パケットを転送するためにはどちらかが使用される。パケットを転送するために実際に使用する経路の選択は、ルーティング技術を使って動的に行われる。効果的なルーティング技術は、ネットワーク全体の性能を大きく影響する。すなわち、ネットワークの遅延を小さくするために可能な限り速く経路決定処理を行うことが要求される。優れたルーティングアルゴリズムは容易にハードウェアに実装されるものである。さらに、経路決定処理は、ネットワークに余分なトラヒックを発生させないためにもネットワーク全体の状態情報を使用すべきでない。ルーティングアルゴリズムは、通常次の3つの属性に基づいて分類される。

1. 経路決定が行われる場所
2. 経路決定が参照する情報の時間定数
3. 動的ルーティングに使用される制御機構

単一のLANにおいては、送信側がメッセージを通信チャネルに単に送り出し、受信側がチャネルからメッセージを受信できるためルーティング技術は要求されない。すなわち、送信側から受信側にメッセージを送信するために経路を決定する必要がない。

経路決定が行われる場所
この属性に基づいて、次の3つにルーティングアルゴリズムは分類できる。

1. 送信元ルーティング (**source routing**):この方式では、発信側コンピュータのPSEがパケットを送信する前に全ての経路を選定する。すなわち、パケットが転送される宛先までのルーティングする全てのPSEはパケットの送信元

コンピュータの PSE で決定され、このルーティング情報はパケットに含まれる。この方式は、送信元コンピュータの PSE がネットワーク環境について十分に情報を待つことを要求する。しかし、ルーティングの PSE は経路決定に何ら関与しないため経路決定処理は効率的である。この方式の欠点は、送信元コンピュータの PSE からパケットが一旦送信されるとその後はルーティング経路を変更できないため、障害に影響されやすいことである。

2. ホップバイホップルーティング **(hop-by-hop routing)**:この方式は、各 PSE が次の PSE への経路のみを決める。すなわち、各 PSE は全ての出力チャネルと隣接の PSE の状態情報を管理し、その中から最適な隣接の PSE を選択してその PSE にパケットを送信する。経路決定は一般にチャネルの利用可能状態と隣接 PSE がパケットを受信してルーティングできる状態に基づいて行われる。この方式は、各 PSE がある順序付けされたルーティングテーブルを持つことを要求する。しかしながら、この方式は静的ルーティング方式に比べてネットワークの帯域をより効果的に使用し、パケットの転送経路の代替えにより障害に対する弾力性が高い。

3. ハイブリッドルーティング **(hybrid routing)**:この方式は、前者 2 つの方式を組み合わせたものであり、送信元コンピュータの PSE が特定のルーティングすべき PSE を決定し、その PSE 間の経路についてはホップバイホップルーティング方式によって決定する。

静的ルーティングと動的ルーティング

経路決定のための情報とその更新頻度によって、ルーティングアルゴリズムは次の 2 つの方式に分類できる。

1. **静的ルーティング (static routing)**:この方式は、ルーティングテーブル (PSE にある) が一度設定されると長期間変更されない。このテーブルは、ネットワークの構成が大きく変わった時に変更される。静的ルーティングは、**固定ルーティング (fixed routing)** 又は**確定ルーティング (deterministic routing)** とも呼ばれる。この方式は、実装が簡単で容易である。しかし、この方式はネットワークの帯域の利用率が低く、別の経路が使用できるにもかかわらずパケットを転送できないことがある。したがって、この静的ルーティング方式は構成要素の障害に対して対応能力が低い。

2. **動的ルーティング (dynamic routing)**:この方法では、ルーティングテー

ブルは比較的頻繁に更新され、ネットワーク環境の細かな変化が反映される。動的ルーティング方式は、**適応ルーティング (adaptive routing)** とも呼ばれ、ネットワークの状態の動的変化、例えば障害の発生やチャネルの輻輳に対応することができる。動的ルーティング方式は、パケットの転送に対して複数の経路選択ができるため、ネットワークの帯域を効率良く使用でき、また障害に対しても適応性が高い。特に、後者の特長はネットワーク規模が大きくなるとそれに比例して構成要素の障害が増加するため大規模なネットワークに対して重要なアーキテクチャである。しかしながら、動的ルーティングでは、メッセージを構成するパケットの到着順序が受信するコンピュータで変わることがある。この問題は、パケットに順序番号を付け、宛先のコンピュータで適切にパケットの順序制御を行うことによって解決できる。

動的ルーティングの経路選択方針は、**最短 (minimal)** 又は**非最短 (non-minimal)** である。最短の方針では、送信元と宛先のコンピュータ間の最短経路から 1 つを選択する。したがって、経由するチャネルは宛先に向けて必ず選択される。一方、非最短の方針では、ネットワークの現状に応じてパケットは最短でない経路を経由することになる。このためこの方針に基づく場合には、パケットが宛先にたどり着かないままルーティングし続ける状況を避けることが必要である。

動的ルーティングのための制御機構

動的ルーティングにおいては、ルーティングテーブルは定期的に更新される。この更新のための制御方法として次の 3 つの中から 1 つが使用される。

1. **単独方式 (isolated manner)**:この方式では、個々の PSE は幾つかの経路を調べまた性能を考えて単独でその装置にあるルーティングテーブルの情報を更新する。

2. **集中方式 (centralized manner)**:この方式では、ネットワーク環境の変化、すなわち接続形態や性能について 1 つの PSE に定期的に報告する。この PSE は、受信した情報に基づいて全体のルーティングテーブルを定期的に更新する。更新されたルーティングテーブルの情報は、その PSE から送信元の PSE(送信元ルーティングの場合) 又は全ての PSE(ホップバイホップルーティングの場合) に定期的に送信される。

この集中方式の主な利点は、経路が全体的に最適化でき、また他の PSE は

ネットワークの状態を知らなくても良いことである。しかしながら、集中方式は大規模なシステムやトラヒック変動が激しい場合には性能的にあまり良くない。また、1つのPSEで集中的にルーティングテーブルを構成するため信頼性が低い。

3. 分散方式 (decentralized manner):集中方式の欠点を補うために幾つかのシステムでは分散制御機構を使用している。この方式では、各PSEはPSE相互の情報交換によってルーティングテーブルを更新する。PSEは、他のPSEに送信するメッセージに付加してルーティングテーブルの更新情報を通知することもある。

2.5 通信プロトコル

本章ではこれまでコンピュータネットワークにおいて通信を行うために必要となる幾つかの取り決めについて述べた。例えば、パケットの送受信者間でパケットヘッダ内の各フィールドの位置と大きさ、実際のデータの位置と大きさ、検査合計フィールドの位置と大きさ、誤り検出のための検査合計の計算方法などを取り決めておくことが重要である。複数のパケットからなるメッセージデータの転送のために、送受信者間で最初のパケットや最後のパケットの識別方法を確認しておく必要があり、また最後のパケットでは一部分しか利用されていないためメッセージの最後のビットを示す方法も必要となる。さらに、重複メッセージの扱い方、バッファオーバフローの避け方、及びメッセージの順序の保証方法などについても取り決めが必要である。通信を行う上で相互に取り決めておくことが必要となる全ての事柄は、ネットワーク設計者によって規則や慣習として定義される。このような規則や慣習の集合は**プロトコル (protocol)** と呼ばれる。

コンピュータネットワークは**階層化プロトコル (layered protocols)** の概念に基づいて実現される。この概念によれば、ネットワークのプロトコルは複数の階層から構成される。各階層は、論理的に表現するとネットワークの他のサイトのピアエンティティとデータの交信及び機能の提供を行うための論理的なプロトコルを持っている。隣接する2つの階層のエンティティは、ヘッダ、トレイラ及びデータのパラメータなどを階層間で引く渡すように定義された共通のインタフェースを介して物理的に相互作用を持つ。ネットワーク設計において階層化プロトコルの概念を導入するのは、次の理由による。

■ネットワークのプロトコルはとても複雑である。階層的にプロトコルを設計することで実装の管理が容易となる。

■プロトコルの階層化により階層間のインタフェースが明確になり、1つの階層の変更が隣接の階層に影響を与えない。すなわち、各種の機能が分解でき、独立に実現できるため、他の階層に影響を与えることなくそれぞれの階層を技術革新にそって変更することができる。例えば、ネットワーク制御プログラムのルーティングアルゴリズムを変更しても、ネットワークアーキテクチャとして他の階層に位置付けられるメッセージ順序機能には影響を与えない。

■プロトコルの階層化は、異なった場所の機能的に同じ階層間の相互動作をもたらす。すなわち、この概念により、遠隔サイトへの機能の分散を可能とする。

プロトコルスウィート (protocol suite)、**プロトコルファミリ (protocol family)** 又は**プロトコルスタック (protocol stack)** の用語は、特定のネットワークシステムにおける (全ての階層の) プロトコルの集合を意味する。

2.5.1　ネットワークシステムのプロトコル

第1章で、分散システムとネットワークシステムの基本的な相違について説明した。したがって、これらの2つのシステムに対する通信プロトコルの要求も異なる。ネットワークシステムの通信プロトコルの基本的な目標は、遠隔のコンピュータとの通信ができ、ユーザが遠隔の資源にアクセスできることである。一方、分散システムの通信プロトコルの基本的な目標は、ユーザの遠隔資源へのアクセスだけではなくそのアクセス方法が透過的にできることである。

ネットワークシステムの標準やプロトコルの幾つかは既に使用できる。しかしながら、分散システムのプロトコルはまだ始まったばかりで、使用できる標準はまだできていない。次に、標準のネットワークプロトコルモデルについて述べる。分散システムのプロトコルについては次の章で説明する。

ISO/OSI 参照モデル

階層の数、各階層の名前、及び各階層の機能はネットワークによって異なる。しかしながら、ネットワーク通信プロトコルの設計者が容易に作業できるために国際標準化機構 (ISO) は7階層の標準を定め、各階層で行う内容を定義する

参照モデルを開発した。このモデルは、**開放型システム間相互接続参照モデル (Open Systems Interconnection Reference Model)**(略して **OSI モデル (OSI model)**) [DePrycker et al. 1993, Larmouth 1993, Stallings 1993c]。これは指針書であり、仕様書ではない。これは、各階層のサービスとプロトコルの標準を作成するための骨組みを提供する。標準プロトコルを定めることは、オープンな分散システムを設計する上でも重要である。標準プロトコルが利用できるのであれば、分散システムの個々のソフトウェアの構成要素が異なったアーキテクチャの (コードの順番やデータの表記が異なる) コンピュータ上で独立に開発することができる。階層化ネットワークプロトコルの構造や機能を理解するために OSI モデルの基本について次に述べる。このモデルの詳細については多くの参考文献がある [Tanenbaum 1988, Stallings 1993c, Larmouth 1993]。

OSI モデルのアーキテクチャを図 2.7 に示す。これは、7 階層のアーキテクチャであり、各階層にプロトコルが定義される。このように、各階層は独立した機能を持ち、1 つ以上の特定の通信機能に関連している。7 階層の役割についてその概要を以下に述べる。

物理層 (physical layer): 物理層は、2 つのサイト間のビット列の伝送を行う。すなわち、バイナリ値の列をケーブル、光ファイバ、回線又はマイクロ波／無線に応じて電気信号、光信号又は電磁信号などに変換する。電気的な詳細事項、例えば値 0 と 1 を何ボルトにするか、単位時間当たり何ビット伝送するか、又は片方向の伝送か双方向同時伝送かについては、物理層のプロトコルによって決まる。さらに、物理層プロトコルは機械的な事柄、例えば接続プラグの大きさや形状、ピンの数、及び各ピンの機能についても規定する。すなわち、物理層のプロトコルは、2 つのサイト間のビット列の伝送に関する機械的規格、電気的規格、手順及び機能特性について定める。RS232-C は、シリアル通信回線の良く知られた物理層の標準である。

データリンク層 (data-link layer): 物理層は単に送信サイトから受信サイトへビットとしてデータを伝送する。データリンク層は、伝送されるデータの誤り検出と訂正を行う。物理層はビット列として認識するのに対して、データリンク層はビット列をフレームに分割し、このフレーム単位に誤り検出や訂正を行う。また、データリンク層は、受信側が処理できる速度より速く送信側がフレームを送ることのないように 2 つのサイト間でフレームのフロー制御を行う。したがって、ネットワークの誤り制御とフロー制御機構は OSI モデルではデータリンク層のプロトコルによって行われる。したがって、データリンク層と物理層のプロトコルは 2 つのサイト間のビットの伝送に関して誤りのない伝

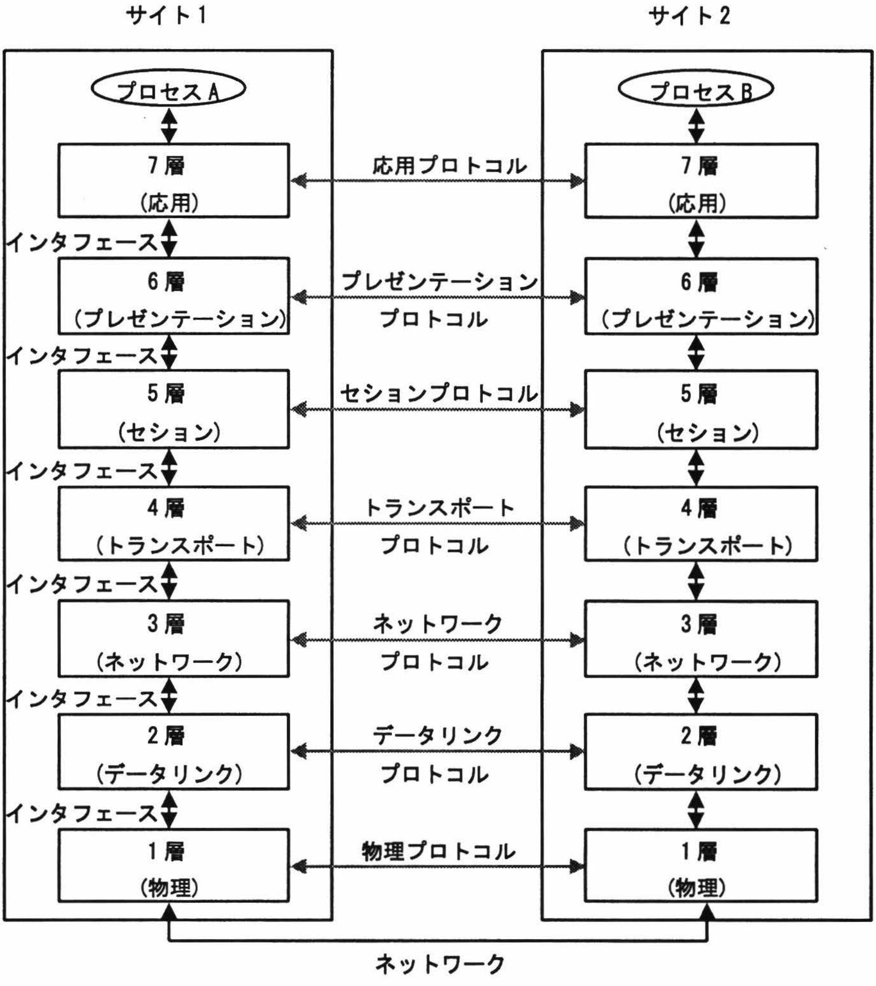


図 **2.7**　OSIモデルにおける層、インタフェース及びプロトコル

送を実現する。

ネットワーク層 (network layer): ネットワーク層は、2つのサイト間の通信のために論理的な通信路を設定する。この層は、上位レベルのアドレスとルーティング方式によりあるサイトから別のサイトにフレームをパケットとして転送する。すなわち、ネットワーク層の主な仕事はルーティングであり、ルーティングアルゴリズムがネットワークにおけるネットワーク層のプロトコルの主な機能となる。

ネットワーク層のプロトコルとしてよく知られているものは、**X.25 プロトコル (X.25 protocol)** と**インターネットプロトコル (Internet Protocol)**(IP と呼ばれる) の2つがある。X.25 は、**コネクション型プロトコル (connection-oriented protocol)** で実際の通信に先立って送信側と受信側の間に仮想回線を確立する概念に基づいている。このプロトコルでは、コネクションの要求がまず宛先に送られ、受理されるか拒否される。コネクションが受理されると、その後の要求のためにコネクション識別子が割り当てられる。コネクション確立フェーズでは、通信するための経路もその後のトラヒックを転送するために決定される。

一方、IP は**コネクションレス型プロトコル (connectionless protocol)** であり、メッセージを送信する前に送信側と受信側の間にコネクションは確立されない。したがって、メッセージの各パケットは独立に転送されるため異なる経路を通ることもある。IP は、DoD(米国国防省) のプロトコルスウィートの一部である。

ネットワーク層で実行される機能は、主に WAN で必要となるものである。単純な LAN では、パケットはネットワークのあるサイトから他のサイトに直接転送できるためネットワーク層は冗長と言える。したがって、ネットワーク層があったとしてもほとんど機能しない。

トランスポート層 (transport layer) :トランスポート層は、サイトツーサイト間の通信機能を提供し、通信サブネットの詳細を上位のセション層に隠蔽するネットワークに依存しないトラスポートサービスを提供する。このサービスを利用することで、通信サブネットの内容には関知することもなくなり、トランスポート層より上位層に影響を与えることなくサブネットを置き換えることができる。特に、トランスポート層はセション層から任意の長さのメッセージを受け付け、パケットに分割した後、ネットワークに転送し、宛先でパケットの再組み立てを行う。パケットは送信側から受信側の間でなくなることもあり、またネットワーク層で使用されるルーティングアルゴリズムに依存するが、

パケットは送られた順序とは異なった順序で宛先に到着することもある。トランスポート層のプロトコルは、このようなパケットの紛失や順序誤りのための制御機構を持っている。このために、トラスポート層は各パケットに順序番号を付与し、パケットの抜けや正しい順番に構成し直すために使用する。

ISOモデルは、誤り制御の機能に応じたトランスポートプロトコルの5つのクラス(TP0からTP4)を提供する。TP0は最も機能の低いものであり、TP4は最も機能の高いものである。下位のネットワーク層の機能に応じてクラスの選択を行う。

トランスポートプロトコルとしてよく知られたものに**転送制御プロトコル(Transport Control Protocol)(TCP)**と**ユーザデータグラムプロトコル(User Datagram protocol)(UDP)**の2つがある。この両者ともDARPAインターネットのDARPAプロトコルスウィートとして実装されている。TCPは、コネクション型トランスポートプロトコルであり、ISOモデルのTP4と同じサービスを提供する。これは論理的なコネクション上での信頼性あるデータ配信を確保するためにエンドツーエンド機能を使用する。このために、パケット順序番号やタイムアウト及び再送を含む応答機能を使用する。

UDPは、コネクションレス型トランスポートプロトコルである。このプロトコルは、パケットの紛失、重複や順序誤りが発生するため信頼性がない。したがって、信頼性のある通信を必要としないアプリケーションにのみUDPを使用すべきである。

セション層(session layer): セション層の目的は、プレゼンテーションエンティティ相互の会話の同期を取り、データの交信の管理を行うための手段を提供することである。また、この層は会話のためのセションを確立する前に通信するエンティティ相互の認証を行う。さらに、この層は特定の会話のタイプ、一方向、両方向相互、又は双方向同時を定め、コネクション要求のメッセージによって会話のセションを起動する。セション層のその他のサービスとして、隔離サービス、会話制御及び優先管理がある。隔離サービスは、送信側のセション層が明確にメッセージのまとまりを引き渡すまで受信側のセション層のバッファにそのメッセージのまとまりを保持するものである。このサービスは、トランザクション(メッセージのまとまりから構成される)が1つの単位になるようなデータベースアプリケーションに有効である。会話制御は、送受信メッセージのために使用されるユーザプリミティブが自由に使える会話セションでは有効である。この場合、ユーザは同じセション上で複数の要求を出し、その応答は要求した順番とは異なる順に返されることもある。会話制御は、要求

の順にしたがって応答を並び替える。優先管理サービスは、普通若しくは重要でないメッセージ以外の重要で時間に厳しいメッセージに対して優先度を与える。セション層には、コネクションレス型の通信はない。

プレゼンテーション層 (presentation layer): この層の目的は、アプリケーション層のエンティティ間の通信においてメッセージの情報の形式が異なる場合でも整合性を保証することである。このために、プレゼンテーション層はメッセージデータに対して次に示す幾つかの変換を行う。

■メッセージは通常構造化された情報を含み、それはプログラミング言語で使用されるデータ型、例えば整数、文字、配列、レコード及びユーザ定義のデータ型などである。したがって、送信元と宛先の言語システムやアプリケーションシステムでデータ型が異なる表現であれば変換が必要となる。

■送信側と受信側のコンピュータのハードウェアが異なるデータ表現を使用する場合もコンピュータ間のデータ転送に対してデータ形式変換が必要となる。この場合、送信側のコンピュータのプレゼンテーション層のソフトウェアは、メッセージデータを転送する前にそのコンピュータで使用されている形式から**外部データ表現 (eXternal Data Representation)(XDR)** と呼ばれる標準ネットワーク表現形式に変換する。受信側のコンピュータのプレゼンテーション層のソフトウェアはネットワーク表現形式のメッセージデータをそのコンピュータの形式に変換する。

■機密又は秘密のデータを扱うアプリケーションに対しては、送信側のコンピュータのプレゼンテーション層のソフトウェアは、セション層にメッセージデータを渡す前に暗号化を行う。受信側では、プレゼンテーション層はアプリケーション層に渡す前に暗号化されたメッセージデータを平文に復号する。

■上記と同様に、メッセージデータ量が大きい場合 (例えばマルチメディアデータ) 又はネットワークが低速であったり負荷が高い場合には、メッセージデータを送受信側のコンピュータのプレゼンテーション層のソフトウェアがそれぞれ圧縮しまた伸長を行う。

アプリケーション層 (application layer): アプリケーション層は、ネットワークのエンドユーザに直接支援するサービスを提供する。確かに、アーキ

テクチャとしてこの層に実現される機能はアプリケーションに特有なものである。各アプリケーションはそれぞれ異なる通信要求を持つため、全てのアプリケーションの要求を満たす決まった標準のアプリケーション層のプロトコルの集合はない。したがって、アプリケーション層は、基本的に各種使用されるアプリケーションに対するいろいろなプロトコルの集合であり、例えば電子メール、ファイル転送、遠隔ログイン、遠隔ジョブ入力や分散データベースのアプリケーションがある。良く知られたアプリケーション層のプロトコルとして、X.400(電子メールプロトコル)、X.500(ディレクトリサーバプロトコル)、FTP(ファイル転送プロトコル)やrlogin(遠隔ログインプロトコル)がある。

これらの7階層の実装を考えると、最初の3階層はほぼハードウェアによって実現され、次の2階層はオペレーティングシステムで、プレゼンテーション層はユーザのアドレス空間に置かれるライブラリのサブルーチンとして、そしてアプリケーション層はユーザプログラムで実現される。

OSIモデルにおけるメッセージ転送の例。 OSIモデルにおける各層の機能を説明するために、メッセージ転送の簡単な例を考えてみる。図2.8に示すように、送信サイトのプロセスがメッセージMを受信サイトのプロセスに送信することを考える。送信サイトのプロセスはメッセージMを作成し、そのマシンのアプリケーション層(7層)にそれを渡す。アプリケーション層のソフトウェアは、Mにヘッダ(H_7)を付与し、階層7と6のインタフェースを介してプレゼンテーション層(6層)にそれを渡す。プレゼンテーション層のソフトウェアは、受け取ったメッセージについてテキストの圧縮、コード変換、暗号化などを行い、ヘッダ(H_6)を付与し、セション層(5層)に渡す。会話のタイプに応じて、セション層のソフトウェアは送信側と受信側の間に会話を確立する。またそのソフトウェアはメッセージの転送方向も取り決める。この層でもヘッダ(H_5)が付与され、トランスポート層(4層)に渡される。トランスポート層のソフトウェアはメッセージをパケットと呼ばれるより小さな単位(M_1とM_2)に分割し、それぞれのパケットにヘッダ(H_4)を付与する。このヘッダにはメッセージパケットの順序番号を含む。これらのパケットはネットワーク層(3層)に渡される。ネットワーク層のソフトウェアは、受け取ったパケットの経路を決定し、パケット転送のための送信側と受信側の間に論理的な経路を設定する。そのソフトウェアは、各パケットにヘッダ(H_3)を付与し、データリンク

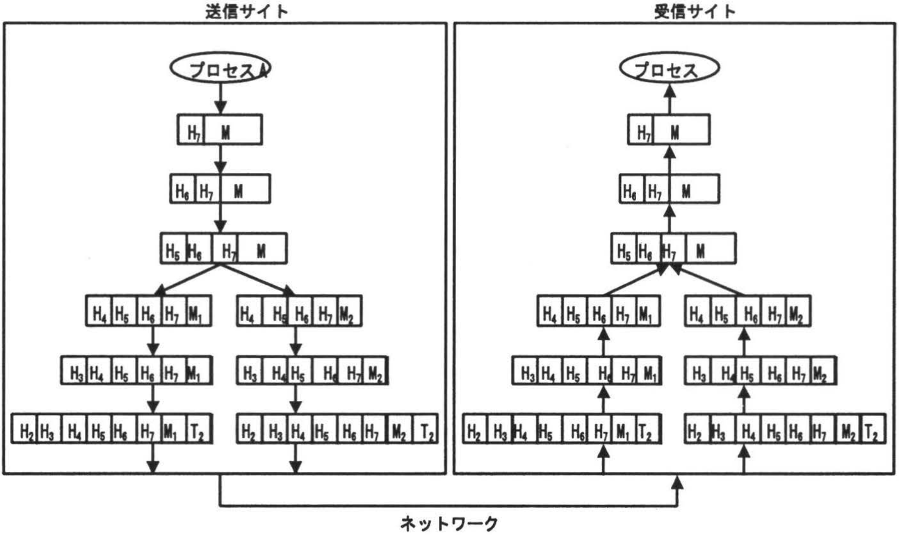


図 2.8　OSI モデルにおける送信サイトから受信サイトへのメッセージ M の転送例、H_n は n 層のヘッダ、T_n は n 層のトレーラ。

層 (2 層) に渡す。データリンク層のソフトウェアは、各パケットにヘッダ (H_2) とトレイラ (T_2) を付与する。このトレイラはパケットのデータの検査合計を含む。このメッセージの単位は、フレームと呼ばれ、物理層 (1 層) に渡される。物理層のソフトウェアは単に送信側のマシンから受信側のマシンに 2 つのマシン間の物理コネクションを使ってビットとして伝送する。

受信側のマシンでは、メッセージデータは物理層からアプリケーション層に順次渡される。メッセージデータが順次上位層に渡されるに従って各層は定められている機能を実行し、送信側で付与されたヘッダやトレイラを取り除く。例えば、受信側のマシンのデータリンク層は各フレームの検査合計を計算値と受信したフレームのトレイラの検査合計と比較し誤り検出を行う。そして、ネットワーク層に渡す前にヘッダ (H_2) とトレイラ (T_2) を取り除く。受信側のマシンのアプリケーション層は、最後に受信サイトの通信プロセスに元のメッセージの形式で渡す。送信側のマシンの各層は、概念的には受信側のマシンの対応する層とのみ通信するが、物理的には送信側のマシンの隣接する層とのみ通信する。この概念がネットワーク設計では重要である。

IEEE802 LAN 参照モデル

ISO モデルは、主に 2 地点間のパケット交換網の環境でのコンピュータネットワークのモデルを主体として考えられているため、LAN よりもむしろ WAN を指向したものである。LAN では、ホストコンピュータは比較的単純なインタフェースハードウェアによってネットワークに直接接続される。各サイトのインタフェースハードウェアやネットワークドライバソフトウェアは、低誤り率でかつ交換遅延のない高速なデータの送受信ができる。LAN のこのような重要な特長は、WAN と比べて価格、速度及び信頼性の点で十分に優れている。LAN と WAN のこのような特徴的な相違のために、OSI モデルと LAN の概念にも下記のような相違があり、OSI モデルは一般的に LAN 環境には不適応と考えられている。

■ OSI モデルでは、2 つの通信エンティティが情報の交信を行うことに合意した後初めて情報の交信を行う。しかし、LAN では、このような事前の合意がなくても通信が行われ、短い時間内に幾つかの送信元から情報が配信される。

■ OSI モデルでは、通信の形態は一般的に 1 対 1 若しくは 1 対多である。しかし、LAN での通信形態は一般的に多対多である。

■ OSI モデルは、コネクション型が主体である。LAN の通信は、ほとんどコネクションレス型である。

以上のことから、LAN への適応を考えると参照モデルの拡張が必要となる。この課題については、既に検討され、LAN に対する参照モデル、**IEEE 802 LAN 参照モデル (IEEE 802 LAN Reference Model)**(略した IEEE 802 LAN) [IEEE 1990] が 1983 年 IEEE で開発された。この IEEE 802 LAN モデルは、LAN のユーザ又は構築者が容易に異なる機種の装置間の相互接続を実現できるように OSI モデルと同様な目的で開発された。従って、IEEE 802 LAN モデルは、OSI モデルの下位 2 層のみを修正し、OSI モデルの上位層プロトコルを利用できるように上位層については修正はない。下位 2 層の修正の主なものは、LAN の物理媒体が全てのサイトに共有されている最も重要な特徴に関連している。IEEE 802 LAN モデルと OSI モデルの関係を図 2.9 に示す。

図に示すように、IEEE 802 LAN モデルの下位 3 層は、物理層、メディアアクセス制御層及び論理リンク制御層である。物理層は、LAN に使用されるベースバンド、ブロードバンド、光ファイバ及びツイストペアの 4 つの媒体のインタフェースプロトコルを定義する。メディアアクセス制御層は、名前が示す



図 **2.9** IEEE802 LAN モデルと OSI モデルの関係

ように、LAN のメディアアクセス制御プロトコル (**medium-access-control protocols**) に関連する。この層は、OSI モデルの物理層とデータリンク層の両者に関係する機能を持つ。この層に、次の 4 つの標準がある。

1. IEEE 802.3 標準、CSMA/CD 方式を使用するバス形態の LAN のプロトコルを規定する。
2. IEEE 802.4 標準、トークンパッシング方式を使用するバス形態の LAN のプロトコルを規定する。このバスに接続されるサイトはトークンパッシング方式を使用する論理リングと同じである。
3. IEEE 802.5 標準、トークンパッシング方式を使用するリング形態の LAN

のプロトコルを規定する。

4. IEEE 802.6 標準、MAN のプロトコルを規定する。

3 層目、すなわち論理リンク制御層は、IEEE 802.2 標準である。この標準は、基本的にメディアアクセス制御層として定義されている 4 つの標準と合わせて使用する共通の論理リンク制御プロトコルを規定する。

OSI モデルで定義されるプロトコルと IEEE 802 LAN モデルで定義される標準との関係については、IEEE 802.1 標準に記述されている。IEEE 802 LAN モデルの詳細については、文献 [IEEE 1990, 1985a, b, c] による。

ネットワーク通信プロトコル事例：インターネットプロトコルのスウィート

ネットワークシステムのためのプロトコルスウィートとして、例えば米国国防省の IP スウィート、ゼロックス社のゼロックスネットワーキングシステム (XNS)、IBM 社のシステムネットワーク体系 (SNA)、拡張ピアツーピアネットワーキング (APPN) 及び NetBIOS がある。この中のプロトコルスウィートとして IP は、魅力的な機能を持っているため最も良く知られ、普及している。例えば、これは LAN や WAN の両者に適用でき、あらゆる機種のコンピュータに実装できるため、ベンダの特定な仕様を含んでいない。これは、ベンダとは独立のインターネットエンジニアリングタスクフォース (IETF) によって開発されたオープンな標準である。全てのベンダは、ネットワーキングにおける共通語として IP を支持している。さらに、IP は優れたネットワーキングの標準であるため、それを基幹プロトコルとして使用する立場をとっている企業は ATM、FDDI、交換型イーサネットやトークンリング、又は 100 Mbps イーサネットのような高速なインターネットワーキング技術へ速やかに移行することができる。

図 2.10 に IP スウィートの構成を示す。これは、5 階層から成り、各層に幾つかのプロトコルがある。1 つの層に複数のプロトコルが存在することにより、各種の通信要求を持つアプリケーションに対してより高い柔軟性を持っている。各層のプロトコルについてその概要を次に述べる。

物理層プロトコル (physical layer protocol): LAN において IP スウィートを使用する多くのシステムは物理層としてイーサネットプロトコルを使用する。また、物理層としてトークンリングプロトコルを使用する IP スウィートに対応した LAN 製品も利用できる。さらに、RS-232 シリアル回線プロトコル

層	各層のプロトコル
アプリケーション	FTP、TFTP、TELNET、SMTP、DNS、その他
トランスポート	TCP、UDP
ネットワーク	IP、ICMP
データリンク	ARP、RARP、その他
物理	SLIP、イーサネット、トークンリング他

図 **2.10** インターネットプロトコルスウィート構造

(1200 bps から 19.2 Kbps の速度) を使用する **SLIP (Serial Line Internet Protocol)**(シリアル回線インターネットプロトコル **(Serial Line Internet Protocol)**) による実装もある。衛星通信やマイクロウェーブの物理層プロトコルを持つ IP スウィートのネットワークも存在する。

データリンク層プロトコル **(data-link layer protocol)**: 全てのプロトコルスウィートは、ネットワークにおいて各コンピュータを識別するための何らかのアドレス方法を定義する。IP スウィートにおいて、コンピュータのアドレス (インターネットアドレス **(Internet address)** 又は **IP** アドレス **(IP address)** と呼ばれる) は次のような情報から構成される。

1. ネット番号 **(net number)** インターネットプロトコルスウィートを使用するネットワークは、特定の管理組織、SRI インターナショナルに置かれるネットワーク情報センター (NIC) によって割り振られたユニークなネット番号を持つ。

2. サブネット番号 **(subnet number)** これは、コンピュータが所属するサブネットを示す付加的な番号である。この番号は、ユーザがサブネットを構成する時に割り付けられる。サブネット番号は、コンピュータがサブネットに所属する時のみコンピュータの IP アドレスとして表示される。

3. ホスト番号 **(host number)** この番号は、ネット番号によって示された

ネットワーク内で一意にコンピュータを識別する。

これらの全ての情報は、4つの8ビットフィールドに分割される32ビットのアドレスで表示される。各フィールドは**オクテット (octet)** と呼ばれる。各オクテットはピリオドによって区切られる。IP アドレスの1バイトを10進数、0〜255の値で表示する。IP アドレスの例として、190.40.232.12となる。

インターネットアドレスには、A、B、Cの3つのクラスとマルチキャスト機能を必要とするアプリケーションのためのインターネットマルチキャスト通信用がある。インターネットアドレスの各クラスの形式と値について図2.11に示す。マルチキャストを利用するアプリケーションの重要性はますます高まっており、ユーザ組織のグループでは **MBone (Multicast Backbone)(マルチキャストバックボーン (Multicast Backbone))** と呼ぶマルチキャストネットワークを構築しており、これは同時に大きなグループに配信する必要があるオーディオ、ビデオ、ホワイトボード及び他の情報ストリームをマルチキャストするインターネットにおける仮想的なネットワークである [Macedonia and Brutzman 1994]。MBone には、広帯域通信、ルーティングプロトコル、データ圧縮及びネットワーク構成などのネットワークの課題がある。多くのアプリケーションツールがホストのプラットフォームを幅広く活用するために利用できる。今日、何百人もの研究者がグループ通信のプロトコルやアプリケーションを開発するために MBone を使用している。

インターネットアドレスの3つのクラスは、組織の大きさに対応するために設計されている。クラス A アドレスは数多くのコンピュータ (ホスト) を含む大規模なネットワークに使用され、クラス C アドレスは少ないホストのネットワーク用であり、クラス B アドレスはその中間的なネットワークに使用される。図2.11から、127のクラス A アドレスのみであり、クラス A ネットワークでの最も値が大きいホストアドレスは255.255.254で、16,777,214のホストを収容できる。一方、クラス C では254ホストの収容となる。

ホストアドレスの部分の各オクテットの値は0から255を取ることができるが、ホストアドレスとして全て0又は全て1の値は使用できない。すなわち、例えば、ネットアドレス78でのホストアドレスは78.15.0.105や78.1.1.255は可能であるが、78.0.0.0や78.255.255.255のアドレスは使用できない。これはホストアドレス値として全て0や全て1は特殊な目的に使われるためである。ホストアドレスが全て0のものは「このホスト」を意味し、全て1のものはネットアドレス内の全てのホストにメッセージをブロードキャストするために

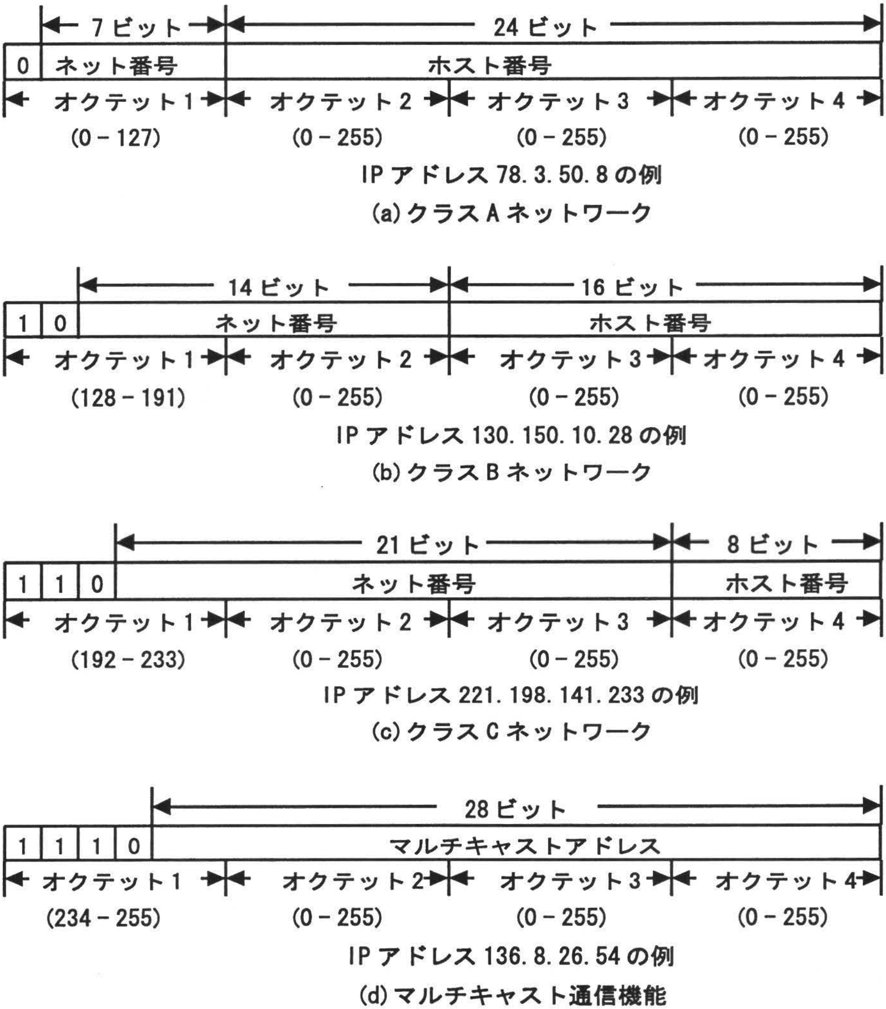


図 **2.11** インターネットアドレス形式

使用される。

マルチホームドホスト (multihomed host) は、複数のネットワークに接続するホストである。これは、複数のインターネットアドレスを持ち、それぞれのネットワークのアドレスである。全てのインターネットアドレスはユニークなホストを表すが、各ホストは1つのユニークなアドレスを持つわけではない。

ネットワーク層からデータリンク層によって受信した各パケットは送信側と受信側のホストのIPアドレスを含む。これらのIPアドレスはネットワークに転送されるために物理ネットワークアドレスに変換されなければならない。同様に、他のネットワークから受信したパケットに組み込まれている物理ネットワークアドレスはネットワーク層に渡す前にIPアドレスに変換しなければならない。IPアドレスから物理ネットワークアドレス、物理ネットワークアドレスからIPアドレスへの変換はデータリンク層で行われる。この層にある重要な2つのプロトコルは、ARPとRARPである。**ARP (Address Resolution Protocol)(アドレス解決プロトコル)** [Plummer 1982] はIPアドレス (32ビット長) からイーサネットアドレス (48ビット長) に対応付けるイーサネットアドレスリュションプロトコルである。一方、**RARP (Reverse Address Resolution Protocol)(逆アドレス解決プロトコル)**[Finlayson et al. 1984] はイーサネットアドレス (48ビット長) からIPアドレス (32ビット長) に対応させる、ARPの逆のIPアドレスリュ解決プロトコルである。

ネットワーク層プロトコル **(network layer protocol)**:ネットワーク層プロトコルとして、**IP (Internet Protocol)** [Postel 1981a] と **ICMP (Internet Control Message Protocol)(インターネット制御メッセージプロトコル)**[Postel 1981b] の2つがある。IPは、あるホストから他のホストにデータグラムの転送を行う。**データグラム (datagram)** は、送受信を行うホストの上位層プロトコルを転送するための情報のかたまりである。これは、送受信するホストのIPアドレスを含む。データグラムの転送におけるIPの2つの主機能は、データグラムのIPパケットからの分割とIPパケットへの組み立て及び各IPパケットのルーティングである。パケットルーティングでは、最少パス選択のためには動的ルーティングアルゴリズムが使用される。

送信ホストにおいて、トランスポート層からデータグラムを受理するとIPは20バイトのヘッダを付加する。このヘッダには多くのパラメータを含み、最も重要なものは送信側ホストと受信側ホストのIPアドレスである。他のパラメータとしてデータブラム長を含み、許容されているネットワークのパケット

長 (**MTU (maximum transfer unit)**(**最大転送単位**) と呼ばれる) をデータグラムが越えているか否かの情報をもち、必要ならば分割される。データグラムがネットワークのパケット長を越えている場合には分割され、それぞれ IP パケットとして転送され、これらの IP パケットは独立に配送することができる。これらの分割された IP パケットは、受信側ホストの IP によって元のデータグラムに組み立てられ、上位プロトコル層に渡される。

IP のデータ転送サービスは、**最善努力転送 (best-effort delivery)** である。すなわち、パケットの転送はできる限り努力して送られるが、保証はされない。IP はヘッダの検査合計を計算し、検証する。ルータでのオーバヘッドを避けるためデータの検査合計は行わないで、上位のプロトコルに委ねている。受信側ホストでは、ヘッダの検査合計の誤りがあればそのパケットを廃棄するため、上位層プロトコルが再送することになる。したがって、IP のデータ転送サービスは信頼性がない。

ネットワーク層の ICMP は、基本的なフロー制御を提供する。IP パケットがホスト又はルータへの転送速度が速すぎて廃棄された場合、ICMP はデータが速く到着したことを送信元に通知するためのメッセージを送り、送信するデータの量を低下するように促す。

トランスポート層プロトコル (transport layer protocol):トランスポート層プロトコルは、ネットワーク上の離れたコンピュータのプロセス間の通信を可能とするものである。この層の 2 つの主要なプロトコルとして **TCP (Transport Control Protocol)**(**転送制御プロトコル**) [Postel 1981c] と **UDP (User Datagram Protocol)**(**ユーザデータグラムプロトコル**)[Postel 1980] がある。これらの 2 つのプロトコルは、ネットワーク層に IP を使用することを示すためにそれぞれ **TCP/IP (Transport Control Protocol/Internet Protocol)**、**UDP/IP (User Datagram Protocol/Internet Protocol)** として記述される。

TCP は、アプリケーションプログラムに対してコネクション型で信頼性があるバイトストリームのサービスを提供する。TCP のコネクションで転送されるデータは、相手に受信されるか、若しくは誤りの指示が与えられるため信頼性のあるプロトコルである。ネットワーク層の IP は信頼性のないコネクションレス型転送サービスを提供するため、TCP はユーザのプロセスに対して信頼性のある仮想回線を提供するための必要なロジックを持つ。したがって、TCP はプロセス間のコネクションの確立と解放、データの順序制御、エンドツーエ

ンドの信頼性 (検査合計、確認通知、タイムアウト制御) 及びエンドツーエンドのフロー制御を行う。ネットワーク層の ICMP は、2 つの通信プロセス間のエンドツーエンドのフロー制御サービスを提供しない。TCP はアプリケーション層で定義される多くのインターネットサービスで使用される。

一方、UDP はコネクションレス型の信頼性のないデータブラムサービスを提供する。これは、障害や紛失が発生しても誤りの指示が与えられないため信頼性のないプロトコルである。したがって、UDP は IP が持っていないプロセスのアドレス付けと UDP データグラムの中身を精査するためのオプション的な検査合計の 2 つの機能を除けば IP に似ている。これらの 2 つの機能は、コネクションレス型のデータブラムプロトコルを要求するユーザプロセスにとって IP を直接使用する代わりに UDP を使用する上で十分である。UDP は実験や通信の信頼性が余り要求されないか、アプリケーションレベルで信頼性が保証されているような小規模な分散アプリケーションでよく使用される。例えば、rwho サービス、ネットワーク監視、時間サービス、簡易ファイル転送プロトコル (TFTP) などで UDP を使用する。

IP 層はホストツーホスト通信サービスを提供し、トランスポート層はプロセスツープロセス通信サービスを提供する。プロセスのアドレス付けのために TCP と UDP とも 16 ビットの整数の**ポート番号 (port number)** を使用する。すなわち、TCP と UDP は転送するデータにヘッダを付ける。このヘッダの中に送信プロセスと受信プロセスを示すためのポート番号を含む。ポート番号は、特定のコンピュータ内で一意に識別できるようにプロセスを示し、かつそのコンピュータ内でのみ正しいものである。IP パケットが宛先のコンピュータに届くと、トランスポート層プロトコルはコンピュータの指定されたポート番号に渡す。ポート番号によって指定されたプロセスは、そのポートからパケットを受け取る。ポートに基づくプロセスアドレス付け機構についての詳細は第 3 章で述べる。

アプリケーション層プロトコル (application layer protocol): IP スウィートのアプリケーション層には各種のプロトコルがある。これらの標準的なアプリケーションプロトコルは、IP スウィートの実装ではほとんど全てが利用可能である。広く使用されるものについて以下に述べる。これらのプロトコルの詳細は [Stallings 1992b] で見ることができる。

1. **ファイル転送プロトコル (File Transfer Protocol)(FTP)** このプロト

コルは、ネットワーク上の遠隔ホストとのファイル転送に使用される。ファイル転送は次のように行われる。

■ユーザはローカルホストで ftp コマンドを実行し、パラメータとして遠隔ホストを指定する。

■ユーザのマシンの FTP クライアントプロセスは TCP を使用する遠隔ホストの FTP サーバプロセスとコネクションを確立する。

■ユーザが遠隔ホストをアクセスできるために名前とパスワードのログインを要求する。

■ログインが行われると、get(遠隔からローカルへのファイル転送) 又は put (ローカルから遠隔へのファイル転送) コマンドによって要求するファイルが転送される。バイナリやテキストファイルが転送できる。ユーザはディレクトリの一覧表示や遠隔マシン間のディレクトリの移動を行うことができる。

2. 簡易ファイル転送プロトコル (Trivial File Transfer Protocol)(TFTP) このプロトコルもネットワーク上の遠隔ホストとのファイル転送を行う。しかしながら、FTP とは異なり UDP(TCP ではない) を使用し、ユーザの認証を検査せず、またディレクトリの一覧やディレクトリ間の移動の機能はない。

3. TELNET このプロトコルは、端末又は端末に関わるプロセスがネットワーク上のホストと通信することを可能にする。すなわち、ユーザは遠隔のホストにセッションを開始するためにローカルのホストに telnet コマンドを入力する。セッションが確立すると、telnet の入力モードになる。このモードでは、ユーザがキーボードから入力した情報は全て遠隔ホストに送られる。入力モードとして遠隔ホストによって文字若しくはラインモードがある。文字モードでは、キーボードから入力された全ての文字は即座に遠隔ホストに送られ処理される。一方、ラインモードでは入力された情報はローカルでエコー処理され、一行単位で遠隔ホストに送信され処理される。FTP と同様 TELNET も TCP を使用する。

4. 簡易メール転送プロトコル (Simple Mail Transfer Protocol)(SMTP) このプロトコルは、ネットワーク上の 2 つのユーザプロセスが TCP コネクションを使用して電子メールを交換することを可能にする。

5. ドメイン名サービス (Domain Name Service)(DNS) アプリケーショ

ンプロトコル、例えばFTP、TFTP、TELNET、SMTPのクライアントプロセスは、ユーザがアクセスしたいリモートのホストを識別するためにユーザからのインターネットアドレス(10進数の形式)を受理するように設計されている。数値に比べれば、シンボリックな名前は人間にとっておぼえ易くかつ使い易い。したがって、インターネットではホストやネットワークをシンボリックな名前、例えばasuvax.eas.asu.eduやeas.asu.eduのような表記方法を採用している。名付けられたエンティティは**ドメイン (domain)**と呼ばれ、シンボリックな名前は**ドメイン名 (domain name)**と呼ばれる。ドメイン名空間は、インターネットを構成する物理的なネットワーク構成には依存しない階層構造となっている。階層構造方式は名前の空間を管理(第10章で詳細に述べる)する上で柔軟性を提供する。

ドメイン名がアプリケーションプロトコル、例えばFTP、TFTP、TELNET、SMTPなどによってパラメータとして受理されると、下位のプロトコルに通信の要求が出される前にインターネットアドレスに変換される。ドメイン名をインターネットアドレスにマッピングする処理はDNSによって行われる。DNSの詳細については第10章で述べる。

2.5.2　分散システムのプロトコル

既に述べたプロトコルは従来のネットワークアプリケーションであるファイル転送やリモートログインなどに対しては適応できるが、分散システムやアプリケーションに対しては適応し難い。これは、ネットワークシステムに比べて分散システムの要求が下記のような特別なものがあるためである[Kaashoek et al. 1993]。

■透過性 (transparency) ネットワークシステムの通信プロトコルは、位置に依存するプロセス識別子(例えば、ノード内で一意に定まるポートアドレス)を使用する。しかしながら、資源の有効利用を行うために分散システムはプロセス移行機能をサポートする。ネットワークシステムの通信プロトコルではプロセス移行機能をサポートすることはプロセスが移動した場合その識別子を変更しなければならないため、困難である。したがって、分散システムの通信プロトコルは、プロセスが移動しても変更が生じない位置に依存しないプロセス識別子を使用しなければならない。

■クライアントサーバ型通信 (client-server-based communication) ネットワークシステムの通信プロトコルは、ネットワーク上の2つのノード間のデータの転送のために通信を入出力機器として扱う。分散システムでの多くの通信はクライアントサーバモデルに基づいており、クライアントはメッセージをサーバに送ることによってある仕事を依頼し、サーバからの応答があるまで待つ。したがって、分散システムでの通信プロトコルは要求と応答の動作を支援する機能を持つ簡単で、コネクションレス型のプロトコルであれば良い。

■グループ通信 (group communication) 幾つかの分散アプリケーションは、1つの送信者からn個の受信者に信頼性のあるメッセージ送信であるグループ通信機能を必要としている。多くのネットワークシステムはデータリンク層でのブロードキャストやマルチキャスト機能を提供しているが、これらの通信プロトコルはアプリケーションに対してこのような有効な機能を見せていない。さらに、ブロードキャストはn個の2地点間のメッセージ転送により、n個の応答を待つため、非効率で帯域を浪費してしまう。したがって、分散システムの通信プロトコルはより柔軟性のあり効率的なグループ通信を実現するために、グループアドレスを1つ又は複数のデータリンクアドレスにマッピングでき、ルーティングプロトコルがマルチキャストアドレスで定義されるグループに所属する全ての受信者にメッセージを転送するためにデータリンクのマルチキャストアドレスを認識できることが重要である。

■セキュリティ (security) セキュリティはネットワークにおいて重要な問題であり、暗号はネットワーク上で転送されるメッセージデータの安全を確保するために共通的に使用される方法である。しかしながら、暗号は使用するためには高価であり、ネットワークの全てのノードや通信チャネルは特定のユーザに対して確実性があるとは言い難い。したがって、暗号は発信ノードから着信ノードに確実性がないノードや通信チャネルを介して通信する場合にのみ使用すべきである。通信プロトコルは、ネットワークのセキュリティが確保されていない場合にメッセージを暗号化するための柔軟でかつ効果的な暗号機構を支援できることが望まれる。ネットワークシステムにおいて既存の通信プロトコルはこのような柔軟性をもつものは少ない。

■ネットワーク管理 (network management) ネットワーク管理、例えばネットワークのノードの追加や削除は、通常ネットワークの最新の構成情報を

反映する構成ファイルを更新するためにシステム管理者の承認が必要となる。例えば、新しいアドレスの割り当ては確認作業を伴う。理想的には、ネットワークプロトコルがネットワークの構成を動的に反映するようにネットワークを自動的に管理するべきであろう。

■規模拡張性 (scalability) 分散システムの通信プロトコルは、規模拡張性に富みかつ LAN 及び WAN 環境において効率的な通信を実現しなければならない。1つの通信プロトコルによって両者のネットワークに対して使用できなければならない。

分散システムにおける高スループットと応答性を実現し、また上記の課題に向けて設計された2つの通信プロトコルとして、**VMTP (Versatile Message Transfer Protocol)**(高機能メッセージ転送プロトコル) と **FLIP (Fast Local Internet Protocol)**(高速ローカルインターネットプロトコル) がある。VMTP は、グループ通信機能を提供し、また安全で効果的なクライアントサーバ型の通信プロトコルである [Cheriton and Williamson 1989]。一方、FLIP は透過性、効果的なクライアントサーバ型通信、グループ通信及び容易なネットワーク管理を実現するように設計されている [Kaashoek et al. 1993]。これらのプロトコルについて次に述べる。

高機能メッセージ転送プロトコル

このプロトコルは、特に分散オペレーティングシステム用として設計され、V-System[Cheriton 1988] で使用されたプロトコルである。これは、クライアントと複数のサーバプロセス間の要求と応答動作を支援するための特別な機能を持つコネクションレス型プロトコルである。これは、クライアントから複数のサーバに送られる要求メッセージとサーバからクライアントに返送される応答メッセージから成るメッセージトランザクションの概念に基づいている。多くのメッセージトランザクションは、単一の要求メッセージと単一の応答メッセージから成る。

性能向上のために、応答は対応する要求の確認通知としての意味を持ち、応答に対する確認通知は通常同じクライアントからの次の要求によって行われる。特別な機能を使用することによって、クライアントは要求に対する即座の確認通知を要求することができ、またサーバも応答に対しての確認通知を要求できる。

透過性の支援やグループ通信機能の提供のために、VMTP のエンティティは

ホストアドレスがユニーク性、規模拡張性及び独立性を保持するために64ビットの識別子で識別される。後者の特徴によってこのエンティティはネットワーク層のアドレス付けからの独立性を持っている。エンティティ識別子は、複数のエンティティから成るグループを識別するエンティティグループ識別子を含む。例えば、個別のエンティティであるファイルサーバはファイルサーバのグループに所属し、単一のエンティティグループ識別子によって識別される。特定のファイルディレクトリの場所を探すために、クライアントはファイルサーバのエンティティグループにこの情報についての要求を送り、ディレクトリを持つサーバから応答を受ける。グループ管理プロトコルは、新しいグループの生成、グループへのメンバーの追加及びメンバーの削除、さらにグループに関する問い合わせの機能を提供する。

さらに性能を向上するために、VMTPは選択再送 (**selective retransmission**) 機構を提供する。メッセージのパケットは、最大16キロバイトのセグメントデータを含むパケットのグループに分割される。データセグメントは、512バイト (最後のセグメントは小さいこともある) から成るセグメントブロックのシーケンスであり、パケットグループ内のセグメントの位置が32ビットマスクによって指定できるようになっている。各パケットは、パケットに含まれるデータセグメントの位置を表すための転送マスクフィールドを含む。パケット当たりの最大ブロック数は、ネットワークの最大パケット長によって決定する。パケットグループが受信されると、個々のパケットの転送マスクは残りのセグメントブロックを示すためのビットマップを得るために相互に論理和 (OR) の演算が行われる。確認通知のパケットは、このビットマップを含み、送信側は正しく受信されていないセグメントブロックのみを選択的に再送する。

VMTPは、レートによるフロー制御機構を使用する。この機構では、パケットグループ内のパケットは受信側での到着率を低下させるためにパケット間ギャップか広げられる。この機構により、クライアントとサーバは必要なパケット間ギャップの時間を明示的に通信するこたができる、既に述べた選択再送要求に基づいて調整を行うことができる。例えば、受信側から返送されたビットマップが他の全てのパケットを再送することを示していれば、送信側はパケット間隔を適切に広げる。次の確認通知のビットマップが4の倍数番目のパケットが受け取れてないことを示していれば、さらに送信側ではパケット間隔を広げる。パケットの受信が正常に行われると、送信側は受信側の最大受信能力で転送するために定期的にパケット間隔を狭める。このように、選択再送は転送速度が速すぎたり、また高速な送信側から低速な受信側にパケットを起こる

ことによってオーバフローし、性能が低下することを示すことによるフィードバック機能を提供する。

VMTPにおけるある最適化は、繰り返し等価操作と繰り返し非等価操作との差別を行うことである。繰り返し等価操作では、何の弊害なく実行が繰り返される。例えば、日時を要求することは典型的な繰り返し等価操作であり、お金をある銀行口座から別の口座に送金することは繰り返し非等価操作である。VMTPでは、サーバはメッセージトランザクションが繰り返し等価操作であったことを示すラベル付けを応答に対して行うことができる。そうすることによって、要求が再送された時にサーバ応答を再び生成できるため、なくなった応答を再送するための調整が不要となる。しかしながら、応答が繰り返し非等価である時にはVMTPはサーバが要求を一度以上実行することを避ける必要がある。

前述の機能に加えて、VMTPは多様な環境に効果的に適用できる付加的な機能を持っている。リアルタイム通信に有効である機能として条件付きメッセージ配送機能がある。この機能では、サーバが即座に処理できる時にのみメッセージを配送することをクライアントが指定できる。付加的な機能は、性能的な問題を生じることなく、特に共通的な処理において、基本機能を拡張できるように綿密に設計されている。

VMTPプロトコルの詳細については[Cheriton 1986, Cheriton and Williamson 1989]を参照されたい。

高速ローカルインターネットプロトコル

このプロトコルは、分散オペレーティングシステムでのコネクションレス型プロトコルである。これは、Amoeba分散システム[Mullender et al. 1990]で使用されている。この主な機能は、透過性、セキュリティ、容易なネットワーク管理、グループ通信機能及び効果的なクライアントサーバ型通信機能である。下記の記述は、[Kaashoek et al. 1993]を参照している。

透過性については、FLIPは位置に依存しない64ビットの識別子によって**ネットワークサービスアクセス点 (network service access point)(NSAP)**と呼ばれるものでエンティティを識別する。インターネットワークのサイトは、複数のNSAP、一般的には各エンティティ(例えば、プロセス)に複数のNSAPを持つことができる。各サイトは、対応するサイトのオペレーティングシステムのソフトウェア層若しくは別の通信装置で実行される**FLIPボックス (FLIP box)**によってインターネットワークに接続される。各FLIPボックス

は、NSAP アドレスをデータリンクアドレス上にマップする基本的には動的なヒントキャッシュであるルーティングテーブルを管理する。FLIP ボックスのルーティングテーブルの NSAP アドレスを動的に登録、削除するために特別なプリミティブが提供されている。エンティティは FLIP ボックスに複数のアドレス (例えば、エンティティ宛のメッセージを受信するためのアドレスやブロードキャストメッセージを受信するためのヌルアドレス) を登録することができる。FLIP は、エンティティを登録するために使用されるプライベートアドレスとエンティティを外部に知らせるためのパブリックアドレス間の一方向のマッピングを使用する。パブリックアドレスからプライベートアドレスを推定できないように一方向暗号機能が使用される。したがって、NSAP の (パブリック) アドレスを知っているエンティティ(それと通信してるため) は、対応するプライベートアドレスを知らないためにそのアドレス宛のメッセージを受信できない。

FLIP メッセージは、NSAP 間を低信頼性で転送される。FLIP メッセージは、$2^{32}-1$ 以下のサイズである。あるネットワークでメッセージ長が大きすぎる場合には、**フラグメント (fragment)** と呼ばれる小さな単位に分割される。フラグメントは一般に 1 つのネットワークパケットに対応する。FLIP の基本機能は、可変長のメッセージを送信元の NSAP から宛先の NSAP にルーティングすることである。パスの選択方針は、接続されているネットワークについての FLIP ボックスのルーティングテーブルにある情報に基づいている。この目的のためにネットワークの重みと安全ビットの 2 つのパラメータが使用される。**ネットワークの重み (network weight)** が小さいとメッセージは転送されてよいことを意味する。ネットワークの重みは、例えばネットワークの物理的特性、帯域や遅延に基づいている。一方、**安全ビット (security bit)** は重要なデータが暗号されないで転送できるかどうかを示す。

FLIP では、パブリックアドレスにメッセージを送信するための 3 種類の呼び出し、flip_unicast、flip_multicast、flip_broadcast が提供されている。これらは、2 地点間及びグループ通信機能を提供している。グループ通信プロトコルは、flip_multicast を多用している。これは、n 個のプロセスが別々のネットワークにあるとしても 1 つのグループとして 1 つの FLIP アドレスによって指定できる利点がある。

FLIP はそれ自身暗号メッセージではないが、メッセージを安全に配送するために次の 2 つの機構を提供している。最初の機構として、送信側はメッセージの重要度を安全ビットで示すことができる。このようなメッセージは信頼性

のあるネットワークにのみ転送される。第二の機構は、FLIPによって信頼性のないネットワークにメッセージを転送する場合には、**非安全ビット (unsafe bit)** を設定してその旨を示す。このビットの設定されたメッセージを受信した場合、安全なルーティング路があるか否かを送信側と確認できる。安全なルーティング路が存在する場合、送信側は安全ビットを設定し、重要なメッセージを暗号化しないで転送する。転送途中で何らかの状況(ネットワーク構成の変更)で信頼性のあるルーティング路が確保できない場合、そのメッセージは**不到達ビット (unreachable bit)** が設定されて送信側に返送される。この場合、送信側は再度暗号化して、安全ビットを立てないでメッセージを送信する。したがって、要求がある時のみ暗号が行われる。

FLIPは、ネットワーク構成の動的な変化を自動的に処理できるため容易なネットワーク管理を提供する。人的な介在を必要とするネットワーク管理の仕事は、信頼性があるネットワークか否かの指定のみである。すなわち、FLIP自身はネットワークが信頼性あるか否かを決定することはできないため、システム管理者がネットワークインタフェースの信頼性があるか否かを判定した結果を信頼している。

FLIPに対しての要望として、広域ネットワークのフルサポートがある。FLIPは小規模なWANについてのサポートは可能であるが、大規模なWANでのWAN通信プロトコルとして使用するには十分とは言えない。FLIPの設計者は、広域の通信はネットワーク層ではなく、上位層で行われるべきである思っているため規模拡張性に無関心である。

FLIPプロトコルの詳細については、[Kaashoek et al. 1993]を参照されたい。

2.6 インターネットワーキング

第1章で、分散システムの2つの重要な機能は規模拡張性とオープン性であることを述べた。この両者の機能とも複数のネットワーク(異なるベンダーのよって提供されるもの又は異なるネットワーク標準に基づくもの)を1つのネットワークに統合することの必要性を示している。このことによって、別々のネットワークのコンピュータ相互間では通信ができなかったものが相互に通信できるようになる。複数のネットワークを相互接続し、1つのネットワークに構成することを**インターネットワーキング (internetworking)** と呼び、結果的にできたネットワークを**インターネットワーク (internetwork)** と呼ぶ。したがって、複数のLANをWANで接続したものはインターネットワークである。

インターネットワークは、一般に形状やプロトコルが異なる幾つかのネットワークセグメントから構成される不均質なネットワークである。例えば、インターネットワークは複数のLANから成り、あるものは多重アクセスバス形態、他はリング形態で、その中でイーサネットやトークンリングがあり、ネットワークのあるセグメントはIPスウィートで、あるものはIBMのSNA(システムネットワーク体系)プロトコルスウィートである。インターネットワーキングは、比較的間連性のない複数のネットワークを1つの動作するシステムにすることができる。すなわち、インターネットワーキングの目的は異なった物理的なネットワークの詳細を隠蔽し、結果的にインターネットワークを単一の統合装置の機能と見せることである。

インターネットワーキングに関する重要な3つの課題は、複数の(恐らく不均質な)ネットワークを1つのネットワークにする方法、ネットワークを接続する伝送媒体、インターネットワークの管理方法である。これらの課題に対して共通的に使われる幾つかの技術を次に述べる。ここで思い起こさなければならない重要なポイントは、インターネットワークのネットワークセグメントが従来の形状やプロトコルとは代わって広く普及している標準を使用して設計されれば、インターネットワーキングの課題を処理することが容易になることである。ネットワークに対して非標準な技術を適用すれば、全体のネットワークの相互接続のために、既存のネットワークに対して多くの問題を引き起こし、もう一度始めからやり直さなければならない。したがって、標準を採用することは、インターネットワーキングの観点からすると大変重要なことである。

2.6.1 相互接続技術

相互接続技術 (interconnection technology) は、異なる技術やプロトコルを持つネットワークの相互接続を可能にする。同じ技術やプロトコルのネットワークを相互接続することは、ネットワークが相互に通信できるため単純である。しかしながら、異なる技術やプロトコルのネットワークを相互接続することは、ネットワークが相互に通信するための共通の参照ポイントを提供するインターネットワーキングの方式が必要となる。この共通の参照ポイントは、ネットワークに共通となる高位レベルのプロトコル、異なる物理的／電気的特性の技術を持つものを相互接続する装置、又は動作環境の相違を無視できるようにするプロトコルかもしれない。最も共通的に使用される方法は、OSIやIPスウィートのような通信モデル上で共通の層間でデータの移動が可能となる共

通の高位レベルプロトコルを使用することである。インターネットワーキングの手段としてブリッジ、ルータ、ブルータ及びゲートウェイはこの方法を使い易くする。次に、これらの手段の長所、短所及びインターネットワーキングでの特定のアプリケーションについて述べる。これらの手段は、同類若しくは異なるネットワークを相互接続して1つのネットワークとする「インターネットワーキング技術」であり、「ブラックボックス」である。

ブリッジ

ブリッジ (bridge) は、OSIモデルの下位2層 (物理層とデータリンク層) で動作する。したがって、3層以上の通信プロトコルが同じであるネットワークを接続するものであり、物理層とデータリンク層のプロトコルは同じ若しくは異なる場合もある。例えば、ブリッジは2つのネットワーク、一方は光ファイバの伝送媒体で他方は同軸ケーブル、また一方はイーサネット技術で他方はトークンリング技術のネットワークを接続する。しかし、通信するために高位レベルのプロトコル (例えば、TCP/IPやXNS) は両者のネットワークとも同じでなければならない。高位レベルプロトコルが同じであることは、ブリッジにおいてネットワークセグメントから他のセグメントに転送する場合 (フレームを単にコピーする) フレームの形式や内容を変えないことを意味する。したがって、ブリッジは高位レベルプロトコルの透過性を持つ。ブリッジは、別のネットワークセグメントが介在していても2つのネットワークセグメント間のデータを転送し、転送されるデータの内容には関与しない。ブリッジにとって、中間のセグメントはルーティングするためにあるとしか認識されない。

ブリッジは、所属するネットワークのセグメント内のトラヒックを抑えてデータをルーティングするために学習やフィルタリングの処理を行うという点では知的な装置である。したがって、ブリッジはネットワークの分割にも有効である。ネットワークセグメントの性能が過剰なネットワークトラヒックによって低下する場合、ブリッジによって2つのネットワークセグメントに分割できる。

ルータ

ルータ (router) は、OSIモデルのネットワーク層で動作する。したがって、ルータは相互接続するネットワークセグメントの形状やアクセスレベルプロトコルには関与しない。ルータはOSIモデルの下位3層を使用するため、ネットワーク層より上位の高位レベルプロトコルが同じものであるネットワークの相互接続を行う。ルータにとってデータリンク層や物理層のプロトコルは透過で

ある。したがって、2つのネットワークセグメントがこれらの層で異なるプロトコルを使用する場合、接続するためにブリッジが必要となる。

ブリッジとは異なり、ルータはエンドからエンドまでのインターネットワークを意識しない。すなわち、ブリッジはデータの最終宛先を知っているが、ルータはネットワーク上を転送されるデータに対して次のルータはどこであるかのみ知っている。しかしながら、ルータはルーティングするためにデータをコピーするだけでなく、ルーティングテーブルの情報を使用して最適な経路を選択するためブリッジよりも高機能である。すなわち、トラヒックの輻輳を管理することはルータの大きな利点であり、輻輳している経路を避けて別の経路にルーティングするフロー制御機構を持っている。

ルータは、同じ通信プロトコルを使用しているネットワークセグメントを相互接続し、大規模なインターネットワークを構築するために使用される。ルータは、知的な経路制御によりトラヒックの流れを制御できるため特に有効である。

インターネットワークでは、ルーティングとマルチプロトコルの問題を解決するためにブリッジとルータの両者を使用する。この要求に応えるためにブルータ (brouter) と呼ばれる装置が設計され、ブリッジとルータのハイブリッド機能を持つ。この装置は、複雑で高価であり、実装が難しいが、同じ高位レベルプロトコルのネットワークセグメントから成る大変複雑な不均質なインターネットワークに対しては最も良いインターネットワーキングの解決方法を提供する。

ゲートウェイ

ゲートウェイ (gateway) は、OSIモデルの上位3層 (セション層、プレゼンテーション層、アプリケーション層) で動作する。これは最も汎用的なインターネットワーキングの手段であり、異なった通信プロトコルを使用するネットワークを相互接続するために使用される。すなわち、ゲートウェイは全体として異なった通信アーキテクチャのネットワークを相互接続するために使用される。例えば、ゲートウェイはIPスウィートのネットワークとSNAプロトコルスウィートのネットワークを相互接続する場合に使用される。

ゲートウェイによって相互接続された個々のネットワークは異なるプロトコルを使用するため、ゲートウェイの主要な仕事はプロトコル変換である。これに加えて、ゲートウェイはルーティング処理を行うこともある。

2.6.2 どのような伝送媒体を使用するか？

インターネットワーキングにおける別の重要な課題は、2つのネットワークに接続に使用すべき伝送媒体を決めることである。これは2つのネットワークの位置や要求される性能に大きく依存する。例えば、米国標準委員会(ANSI)で規定されている **FDDI (Fiber Distributed Data Interface)(ファイバ分散データインタフェース)** は100 Mbpsの速度で、隣接するビルやキャンパスの複数の建物のLANを相互接続するには理想的な広帯域なバックボーンである。

2つのネットワークがそれほど離れていない(同じ町や、隣の町)、またその間のデータトラヒックが大きくなければ電話回線でつなぐこともできる。2つのネットワーク間で重要なデータを頻繁に交信する場合には専用線を使用すれば良い。

最後に、2つのネットワークが遠く離れている(海外や国内の別の都市)場合、公衆データネットワークの通信回線、例えば電話回線や通信衛星が相互接続のために使用される。長距離通信回線は、高価である。さらに、公衆データネットワークの通信回線で2つのネットワークを接続すると、トラヒックの不整合やシステムの信頼性の面で2つのネットワーク間の性能に影響を与える。セキュリティについてもこの場合問題となる。性能の問題は、トラヒックのルーティングを別の方法で対応することで解決することができるかもしれない。例えば、2つのネットワークがニューヨークとロスアンジェルスにあった場合、全体のトラヒックは性能を上げるためにデンバーとダラスを介してルーティングされるかもしれない。セキュリティの問題については第11章で述べる。

2.6.3 ネットワーク管理技術

ネットワーク管理 (network management) は、ネットワークの状態や動作を監視し、分析することである。ネットワーク監視のツールはネットワークセグメントの状態を監視し、性能やノード及びリンクの障害に関する情報、さらにはネットワーク管理者に有効と思えるネットワークの全体的な現象に関する情報を提供する。単純なネットワーク監視のツールは、何もなければネットワークに問題が発生していないことを報告する。一方、ネットワーク分析のツールはパケットレベルプロトコルの分析を含む幅広い角度からネットワークの動作を分析する。これは、ネットワークの動作について幅広い情報を提供することによって監視による定性的なデータに加えて定量的な情報を与える。

インターネットワークの管理は、幾つかのLANを1つに統合するために個別の問題が全体的な問題になるため単一の独立したLANの管理よりも複雑になる。インターネットワークにおいて問題の場所を局所化することは、問題がローカルのLANか遠隔のLANかとなるため余り有効的ではない。したがって、単一の独立したLANの管理のためのツールは、インターネットワークに適用する場合余り有効でないか、若しくは全く役に立たない。インターネットワークの不均質性は、インターネットワークの汎用的な管理手段を設計する上で大きな障害となる。全ての管理者は、次の機能を持つ単純な手段を望む。

1. 典型的な不均質なインターネットワークに使用されるプロトコルや機器に対応できる。
2. 簡単に使用できる。応答性の良く、ネットワークの分析能力を持たなくても利用できるグラフィカルなユーザインターフェースである。
3. 学習機能やネットワーク障害の診断分析ができる。
4. 特定の機器のベンダーやプロトコルに依存しない。

現在のところ、このようなツールは存在しない。しかしながら、将来を考えるとネットワーク管理フレームワークやネットワーク管理プロファイルはこの方向に向っている。将来のマルチベンダのネットワークの相互運用管理を提供するために、OSI、IETF、OSF及びその他の組織では参照モデルやネットワーク管理フレームワークに基づいて通信ネットワーク管理の標準を開発している。現れる多くの中で、将来のネットワーク管理として3つの標準が有力である。これらは、**簡易ネットワーク管理プロトコル(Simple Network Management Protocol)**(SNMP)、**共通管理情報プロトコル (Common Management Information Protocol)**(CMIP) 及び**分散管理環境 (Distributed Management Environment)**(DME) である。

SNMPは、クライアントサーバモデルでリアルタイムな障害通知を要求しないアプリケーションに適したクライアントサーバプロトコルである。これは、IPスウィートのネットワークを制御監視するために1980年代の後半に現れた。実装の容易性と低価格から、SNMPに基づくツールはほとんどのネットワーク管理のベンダーによって提供されている。速度、セキュリティ及びマネージャ-マネージャ通信機能に関して、IETFでSNMP第2版のプロトコルが検討されている。SNMPの詳細については [Datapro 1990, Shevenell 1994, Janet 1993] を参照されたい。

CMIPは、OSI(ISO/CCITT)において開発されたネットワーク管理の標準である。これは、大規模でかつマルチベンダー環境でのネットワーク管理製品とサービスに対して相互運用性と真の統合化を実現するために設計されている。これは、マネージャ-エージェントモデルに基づいている。管理するシステムは管理動作を呼び出し、管理されるシステムはマネージャに通知する。管理するシステム間の通信は、エージェント-マネージャの方式で行われる。CMIPの詳細については[IT 1990, Janet 1993]を参照されたい。

ネットワーク管理ツールとしては、CMIPに基づく製品よりもSNMPに基づく製品の方が多くある。SNMPのものよりも高機能にもかかわらず、CMIPの製品は高価で、複雑で、処理能力が要求されるため、急速には広まっていない。

OSFのDMEは、分散ネットワーク管理製品のための仕様である。この目的は、マルチベンダー環境における一貫性のあるシステムとネットワークの管理方式を提供することにある。この目的を達成するために、DMEはSNMP、CMIP及び他の業界標準に基づいて設計されている。DMEに基づく製品は、まだ市場には出ていない。DMEの詳細については[Datapro 1993]を参照されたい。

2.6.4　インターネットワークの事例：インターネット

インターネット (Internet) は、インターネットワークの最も良い事例である。これは、ホストコンピュータの統一的なアドレス付けや整合あるプロトコルスウィートによって相互接続された不均質なネットワークの集合体として単一な全世界規模のネットワークである。インターネットのホストや資源は、第10章で説明されるDNS(ドメイン名システム)名前方式によって名前付けされる。インターネットは、米国国防省の高度研究プロジェクトのARPAnetシステムが元となっている。ARPAnetは、最初のWANであり、1969年4サイトから始まった。ARPAnetの基本方式から発展したインターネットは、最初は研究機関や大学で情報の共有や交換を行うために使用された。1989年に商用化に移行することで急速に発展し、今では10000以上のネットワークがつながり、世界の150以上の国で3百万以上のコンピュータ及び4千万のコンピュータユーザがいる。インターネットは今も急成長している。

インターネットは、多くのユーザの興味を引く膨大な情報のネットワークである。インターネットでユーザが目的とする情報を見つけるための幾つかの使い易いツールがある。これらのツールの中で良く知られているもの

に、Gopher、Archie、WAIS(広域情報サーバ)、WWW(World-Wide Web) 及び Mosaic がある。

Gopher [Martin 1993] は、インターネット上の情報を階層的に整理したテキストベースのツールである。これはエンドレスの迷路のディレクトリメニューに似ている。1991 年にミネソタ大学で開発され、現在インターネットの情報検索ツールとして使用されている。詳細については、ftp の boombox.micro.umn.edu のディレクトリ/pub/gopher/docs を参照されたい。

Archie は、インターネットの匿名 ftp によって公開のサーバのファイルインデックスの検索を行うツールである。詳細については、Gopher で gopher.gmu.edu を参照されたい。

広域情報サーバ (Wide-Area Information Servers)(WAIS)、インターネットの情報検索のツールであり、フリーウェア、シェアウェア又は商用のソフトウェアプログラムの 1 つである。詳細については、Gopher で gopher.gmu.edu を参照されたい。

World-Wide Web (WWW) はハイパーテキストを他のテキストドキュメントやファイルにリンクするためのハイパーメディア分散システムである。この機能によってユーザは特定の用語をクリックすることで、その用語についてのより詳細な情報を知ることができる。WWW によって、ユーザは世界中の有効な情報を提供するサイトの静止画、イメージ、動画像などにアクセスできる。WWW は、1992 年に分散ハイパーメディアシステムを構築するために欧州核研究センター (CERN) によって研究された。インターネットにおいて WWW のトラヒックは急速に増大しており、商用インターネットでは良く利用されている。詳細については、[Vetter et al. 1994] 又は Gopher で info.cern.ch を参照されたい。

Mosaic は、インターネットから情報を見つけ出したり、検索するためのハイパーメディアベースのブラウザのツールである。Mosaic のブラウザは、X Windows が動作する UNIX ワークステーション、マイクロソフト Windows が動作する PC、アップルマッキントッシュコンピュータで利用できる。モザイクは、WWW サーバ、WAIS、Gopher サーバ、Archie サーバなどのデータをアクセスする。この機能の有効性や能力から急速に普及している。詳細については、[Vetter et al. 1994] 又は匿名 ftp で ftp.NCSA.uiuc.edu のディレクトリ/PC/Mosaic を参照されたい。

インターネットの世界規模のサービスは非営利及び商用の両者によって多くの有効な方法で使用できる最も利用できるツールとして提供されている。イン

ターネットの現在最も利用されているものをここに示す。以下に示すものは、[Nejmeh 1994] の内容に基づいている。

1. **オンライン通信 (on-line communication)**:インターネットの電子メールサービス (e-mail として知られている) は世界中を相互に通信できるように今日ではコンピュータユーザに幅広く使われている。この機能によってインターネットは数百のユーザに対して迅速にかつ生産性の高い通信手段となった。手紙、電話及びファックスに比べ、電子メールは (a) 手紙より速い、(b) 電話と異なり同時にお互いが話しをする必要がない、(c) ファックスとは違って電子メールの内容はコンピュータに蓄積され、容易に編集ができることから多くから支持されている。

2. **ソフトウェア共有 (software sharing)**:インターネットは、多くのシェアウェアのソフトウェア開発ツールと機能にアクセスすることを可能にしている。このようなシェアウェアのツールの例として C++コンパイラ、コードライブラリ、メールサーバ及びオペレーティングシステム (ftp により sunsite.unc.edu から全て利用できる) がある。フリーソフトウェアファンデーションは豊富な GNU ソフトウェア (詳細は匿名 ftp で prep.ai.mit.edu のディレクトリ/pub/GNU を参照) を提供している。

3. **共通に興味のあるトピックスでの情報交換**:インターネットは、多くの**ニュースグループ (news group)** を持っている。各ニュースグループは共通に興味のあるトピックスについての情報を交換する。例えば、ニュースグループ comp.os.os2.advocacy は OS/2 のオペレーティングシステムについて自由な意見交換を行っている。

4. **一般関連情報の通知**:インターネットは、世界中の人々に興味ある情報を通知するための大規模な**電子掲示板 (electric bulletin)** として幅広く使用される。このような情報には、キャリアアップの情報、会議や行事の開催案内、会議や刊行物への投稿案内を含んでいる。

5. 製品の促進:幾つかの会社は、製品の販売促進のためにインターネットサービスを利用している。このような会社では、ftp、Gopher、WWW サーバのサイトを利用して会社の出来事、製品の発表、最近の戦略的企業活動、広報及

び顧客に関心のある各種の情報を適切に広く公開している。例えば、com.sys.sun.announce のニュースグループはサンマイクロシステムズ社の最近の製品情報を流している。

6. **製品についてのフィードバック**:製品の販売促進に加えて、会社では既存製品の顧客の反響、新製品の市場性、これからの製品についての考え方の情報収集にインターネットを使用している。これはインターネットの WWW や Gopher サイトの相互的な調査アプリケーションによって行われている。

7. **顧客支援サービス**:多くのソフトウェア会社は、先例のないレベルの顧客支援を適切に提供するためにインターネットを利用している。インターネットの電子メール、ftp 及び他のサービスを組み合わせて迅速な顧客支援を実現している。例えば、ソフトウェア製品のバグは電子メールで通知され、製品の改良情報、マイナーリリース、状況、問題や制限及び一般的な助言は ftp サーバによって顧客に提供される。

8. **オンライン刊行物と雑誌**:インターネットは、無料若しくは低価格のために数百人もの電子的な購読者を持っている。インターネットの多くの Gopher や WWW サイトには多くの刊行物や雑誌の電子情報がある。例えば、Dow Jones News/Retrieval はインターネットで Wall Street Journal の情報を提供している。多くの研究者がインターネット上に電子図書を構築する方向にある。

9. **オンラインショッピング (on-line shopping)**:インターネットは、仮想商店からなる新しいマーケットの概念をもたらしている。これらの商店は 24 時間 1 年中開店しており、世界中からアクセスできる。これらは ftp、Gopher および WWW サーバによって販売する製品やサービスの情報を提供している。インターネットサービスを利用して顧客は製品の問い合わせや引き合いを出す。定まった認証や証明方法によって、インターネットサービスは顧客からの注文を受理し、支払い方法を確認し、注文を処理する。例えば、インターネットモールは商店の集まりであり、それぞれ製品やサービスを提供する。インターネットモールの製品やサービスの一覧は、ftp により ftp.netcom.com のディレクトリ/pubs/Guides で見ることができる。

10. **ワールドワイドビデオ会議**:ワールドワイドビデオ会議は、1 つの部屋で

会議を行うように各所にいる人達が打ち合わせを行うことのできるインターネットのサービスである。コーネル大学で開発された CU-SeeMe システムは、**インターネットによるビデオ会議 (Worldwide video conferencing)** システムの例である。CU-SeeMe に関する情報は、ftp により gated.cornell.edu のディレクトリ/pub/video/CU-SeeMe.FAQ.7-6.txt を参照されたい。

2.7　ATM 技術

非同期転送モード (Asynchronous Transfer Mode)(ATM) は、将来のコンピュータネットワーキングに必要となるものとしてしばしば引用される。ATM 技術は、音声、映像、データを含む各種のトラヒックを同時に転送するために固定長の短いパケット (**セル (cell)** と呼ばれる) による高速コネクション型交換／多重技術である。情報は共通のクロックを持たないで独立に送られる非同期転送である。この技術について次に概要を述べる。ATM 技術のより詳細な内容については、[DePrycker 1993, Newman 1994, Fischer et al. 1994, Haendel et al. 1994, Vetter 1995, Kim and Wang 1995] を参照されたい。

2.7.1　ATM 技術の主要機能

ATM 技術は、次のような魅力ある機能により将来の分散システムに大きな影響を与えるものと期待されている。

1. ATM 技術は、155 Mbps、622 Mbps さらには 2.5 Gbps のデータ伝送速度を提供することによって広帯域な分散アプリケーションを可能にする。この機能は、幾つかの新しい分散アプリケーション、例えばビデオオンデマンド技術に基づくアプリケーション、ビデオ会議アプリケーション及びマルチメディアデータのデータベース遠隔のアクセスを必要とするアプリケーションを実現する。

2. ATM 技術は、LAN や広域ネットワークの両者に対して高速な伝送速度とサービスを提供し、以前では LAN の環境でしか扱えなかった強力な分散アプリケーションを WAN の環境においても同じように扱えることを可能にする。

3. ATM 技術は、単一の統合した交換機構 (**セル交換 (cell switching)** と呼

ばれる) によって交換 (回線交換とパケット交換) に対する基盤となる方式を提供する。この機能によって、**固定ビットレート (constant-bit-rate)(CBR)** のトラヒックを生成する分散アプリケーションと**可変ビットレート (variable-bit-rate)(VBR)** のトラヒックを生成する分散アプリケーションの両者に適用できる。例えば、映像やデジタル音声などのアプリケーションは CBR のトラヒックを生成する。固定ビットレートのトラヒックは、スループットとサービスレベルが保証されることを要求する。一方、多くのアプリケーションは VBR のトラヒックを生成する。可変ビットレートのトラヒックは、遅延やスループットの変動に対して許容できるものである。

4. ATM 技術は、2 地点間のトラヒックを転送するために仮想ネットワーキングの概念を使用する。この概念によって、複数のアプリケーション間で物理チャネルの帯域を有効に共有することで、複数のアプリケーションが同時に別々の伝送速度で通信することを可能にする。すなわち、各種のアプリケーションに対して全体の帯域を有効に分割して使用することができる。

5. 1 つの送信者と 1 つの受信者からなる 2 地点間通信に加えて、ATM 技術は 1 つの送信者と複数の受信者に対応できるマルチキャスト機能を容易に提供できる。このような機能は、同時に多くの家庭で見るブロードキャストテレビに必要となる。多くの分散協調アプリケーションも、このような機能をしばしば必要とする。

6. ATM 技術は、テキスト、音声、映像、ブロードキャストテレビなどの幅広いマルチメディアデータを効率良く転送するための単一のネットワークの構築を可能にする。したがって、音声のための電話網、データのための X.25 ネットワーク、映像のためのケーブルテレビネットワークのように個別のネットワークは、音声、映像、データ及び他の情報を統合できる単一の ATM ネットワークによって置き換えることができる。この統合化によって費用の低減や通信ネットワークの設計の単純化をもたらす。

7. ATM 技術は、帯域を制御する上で柔軟性がある。すなわち、要求に応じて帯域を提供し、セル単位 (恐らく多くは ATM の速度や転送能力からするとギガセル単位) のネットワーク利用を可能にする。ユーザは、必要に応じてネットワークの帯域を大きくもしくは小さく獲得し、それに応じて費用をはらう。

8. ATM技術は、規模の拡張に対応できる技術である。帯域やデータ速度を自由に調整するために制御信号を使用するアーキテクチャになっている。さらに、LANやWANの両者の環境において全ての情報タイプ(音声、映像、データ)の伝送について同じ交換技術(セル交換)と同じセル形式を用いる。

9. ATM技術は、標準に準拠している。この規格は、国際電気通信連合(ITU)(旧CCITT)に採用され、広帯域統合デジタルサービスネットワーク(B-ISDN)の基礎になるものとして国際的に標準化されている。

2.7.2　ATM技術の基本概念

ネットワークトラヒックの基本的な2つのタイプとしてCBRとVBRがある。CBRのトラヒック(映像やデジタル映像情報から構成)は一定に転送され、VBRのトラヒック(データ情報から構成)はバースト的である。CBRのトラヒックは、低いが固定の帯域を要求し、スループットとサービスレベルを保証する。一方、VBRのトラヒックは短い間隔でかつランダムに広い帯域を要求し、遅延やスループットの変動に対応できる。この2つのタイプのトラヒック特性の違いから、CBRのトラヒックを扱うために回線交換が最適なネットワーキング技術であり、パケット交換がVBRのトラヒックを扱うための最適なネットワーキング技術である。しかしながら、回線交換とパケット交換とも両者のネットワークトラヒックのクラスを扱うには適していない。したがって、ITUの標準機関は複数の伝送速度に対応でき、かつ効果的に低価格な方法として統一的な多重化と交換機構について検討を行い、セル交換と呼ばれるハイブリッド方式の交換技術を発案した。ATM技術は、このセル交換技術に基づいている。

セル交換技術は、デジタルのパケット交換技術を基本としており、このパケット交換技術はパケット内に含まれるアドレスによって仮想パス上でトラヒックをルーティング及び交換する(この方式は、アドレスによってではなく通信の開始時に確立された物理パスにトラヒックをルーティング交換する回線交換技術とは異なる)。しかしながら、X.25やフレームリレーなどのパケット交換技術とは異なり、セル交換技術はセル **(cell)** と呼ばれる短い固定長のパケットを使用する。ATMでは、セルは53バイト長である。ATM技術は、5バイトのヘッダ(アドレスを含む)と48バイトの情報フィールドからなる。

このセルの53バイトのサイズは、音声、映像及び伝送間隔や遅延の変動を大

きくすることができない他のリアルタイムトラヒックと同様にデータを ATM 上で効果的に伝送するために選択された。実際のデータ発生源では多くの場合 64 キロバイトくらいまでの長メッセージを生成する。このようなメッセージを短いセルに分割することによって、ATM は通信チャネルをアクセスする前に 53 バイトセル伝送時間 (155 Mbps の伝送速度で 3 マイクロ秒) 以上の待ちを生じないような音声や映像の CBR のトラヒックに対応できる。フレームによるパケット交換技術では待ち時間は、ランダムとなり、恐らく数ミリ秒になる。短い固定長のセルを使用することで、長いパケットが短いパケットの伝送を遅らせることを回避できる。セル交換では、各セルの伝送毎に次のセル (別のメッセージに属するセル) を伝送できる。

ATM における適切なセルの大きさは、標準委員会での議論の的であった。ATM 技術は、電話関係の人々にとっては音声パケットとして遅延を小さくするために短いセルを要求し、一方データ通信に関心がある人々はメッセージの分割と組み立て時間のオーバヘッドを小さくするために大きなセルを望んだ。長い議論の末、セルの大きさは 2 つの案、32 バイトセルと 64 バイトセルに絞られた。その後の妥協の結果、ITU は 48 バイトにヘッダを加えたセルサイズを決めた。

53 バイトの固定長のセルでは、ATM 技術として CBR のトラヒックである音声や映像などのアプリケーションにもファイル転送などの VBR のトラヒックのアプリケーションに対しても理想的な選択ではない。しかしながら、単一の統合ネットワークとして両タイプのトラヒックを扱うためには最良の技術である。この高速でハードウェアによる交換のため、CBR のトラヒックを通常扱うことが可能である。また、パケット技術であるために、VBR のトラヒックも効果的に扱うことができる。

ATM は、送信側はまず受信側とコネクションを確立するためコネクション型の技術である。しかしながら、送信側と受信側の間で物理的な回線が確立され、通信中それが保持される回線交換とは異なり、ATM 技術では仮想回線が送信側と受信側の間に設定される。すなわち、ATM では他の利用を排除するようなパスの予約は行わない。割りるてられたユーザがチャネルを利用していないときは、他のユーザが自由にそれをつかうことができる。ATM におけるコネクションの確立では、送信側から受信側への経路が決まり、経路情報がコネクション確立に伴って交換機に設定される。送信側から受信側へのメッセージの全てのセルは、交換機の仮想パス上に転送される。コネクションが必要でなくなると、解放され、このコネクションに関する経路情報は交換機から削除

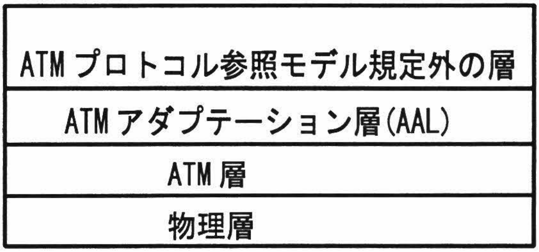


図 **2.12**　ATMプロトコル参照モデル

される。

各セルのヘッダにあるアドレス情報は、セルが転送される仮想パスを決定するためのルーティングプロトコルによって使用される。ATMのアドレスはローカルに意味を持ち、隣接する2つのATM装置間にのみ関係する。仮想パスが確立すると、各交換機にはヘッダアドレスによって入力するセルを識別するためのテーブルが設定され、そのテーブルを参照して適切な出力ポートを定め、新たなアドレスを上書きし、次の交換機がそれを参照する。メッセージは予め決められた経路上の交換機をルーティングされ、その経路は仮想のものである。

ATMでは、仮想パスは多重化できる仮想チャネルのまとまりである。したがって、1つの仮想パス上で2つのホストは多くの個別のアプリケーションのセルを多重化できる。セルは統計多重されるため、各種の分散アプリケーションに動的に可能な帯域を割り当てることができる。これは、予想されるトラヒックや必要となるネットワーク資源の割り当てにしたがって仮想チャネルパスの選択によって行われる。帯域の保証を必要とするアプリケーションに対しては必要となる帯域を指定することが要求される。これは、ネットワークがより効果的に動作するためのATMサービスの仮想的な特性を指定している。

2.7.3　ATMプロトコル参照モデル

ATMのプロトコル参照モデルは、3つの階層、物理層、ATM層及びATMアダプテーション層(AAL)からなる(図2.12)。データ、音声、映像を呼び出すアプリケーションは、これらの3層の上位に位置する。これらの3階層の機能について次に述べる。

物理層

物理層 (physical layer) は、ATM プロトコルスウィートの最下位に位置する。この層は、回線へのビットの送出と受信を行う。この層は、物理媒体依存 (PMD) 副層と伝送コンバージェンス (TC) 副層の 2 つの副層からなる。PMD 副層は、使用する物理伝送媒体 (電気／光) 上のトラヒック伝送の実際の伝送速度を定義する。一方、TC 副層は PMD 副層によって定義される物理伝送媒体上での送信及び受信におけるセルデータと電気／光の搬送波の信号間のプロトコルを定義する。TC 副層のプロトコルは、使用する物理伝送媒体によって異なる。

物理層は、下記の 2 つの方法であるユーザから別のユーザにセルを伝送できる。

1. **同期データストリームとしてのセル伝送:**この場合、ユーザ-ネットワークインタフェース (UNI) は、コンピュータの ATM アダプタボードの形態をとり、回線又は光ファイバにセルのストリームを送出する。伝送ストリームは連続していなければならない、そのため送るデータがない場合には空のセルを伝送する。
2. **他標準のフレーム伝送構造のペイロード部分でのセル伝送:**この場合、UNI は物理層のフレーム化や同期のために標準の伝送構造を使用する。**SONET (Synchronous Optical NETwork)**(同期光ネットワーク)[Omidyar and Aldridge 1993] は、この目的に最も使用されている標準であり、次に概要を述べる。SONET 形式は、現在シングルモード及びマルチモードの光ファイバとより対線でサポートされている。

SONET は、光インターフェース上でデジタル情報を伝送するための国際標準のスウィートである。SONET において、データ伝送の基本単位は図 2.13 に示す構造を持つフレームである。図に示すように、SONET のフレームは 27 バイト (9 × 3) のオーバヘッドと 783 バイト (9 × 87) のペイロードからなり、全体で 810 バイト (9 × 90) である。オーバヘッドのバイトは、誤り監視、システム管理機能、同期制御、ペイロード型の識別に使用される。ペイロードは、例えば T1 信号、T3 信号又は ATM 仮想回線などの各種の信号を伝送する。T1 は 1.544 Mbps の基本伝送速度のデジタル伝送サービスであり、T3 は 28 の T1 回線に相当する 44.736 Mbps の基本伝送速度のデジタル伝送サービスである。バイトの伝送順番は、列の順で、左から右に送られ、1 つのフレームは 125 マイクロ秒で伝送される。1 フレーム 125 マイクロ秒の基本時間単位で、電話シ



図 **2.13**　SONET フレーム形式

ステムの標準サンプリングで1秒当たり800サンプリングとなる。したがって、SONET フレーム形式では、全体のデータ速度は 51.840 Mbps(オーバヘッドを含む) となり、実行データ速度は 49.920 Mbps(オーバヘッドを除く) である。

伝送速度 51.840 Mbps の SONET の基本単位は、**STS-1(Synchronous Transport Signal Level 1)(同期転送信号レベル1)** と呼ばれる。より高速な SONET 信号は STS-n 信号を得るために n フレームの STS-1 をバイトインタリーブする [Vetter 1995]。

STS は光信号よりもむしろ電気信号を使用する。**光搬送波 (Optical Carrier)**(OC) レベルは、スクランブリング (0と1の長い信号列を避け、受信側でクロックの回復を可能にする) され、電気信号からから光信号に変換された後に STS レベルから得られる [Vetter 1995]。このように、OC-n レベル信号は STS-n レベル信号のスクランブリングと変換によって得られる。n として使用される値は 1、3、12 であり、それぞれ 51.84、155.52、622.08 Mbps の伝送速度をもつ OC-1、OC-3、OC-12 の信号レベルである。

ヨーロッパでは、**SDH (Synchronous Digital Hierarchy)(同期デジタルハイアラキ (Synchronous Digital Hierarchy))**[Omidyar and Aldridge

1993] と呼ばれる別の標準フレーム形式が使用されている。SDH のフレーム形式では、基本単位のデータ速度は、SONET の 51.85 Mbps に代わって、155.52 Mbps である。

ATM 層

ATM 層 (ATM layer) は、セルの処理及びルーティングを行う。これらの処理は、セルヘッダの作成、個々のコネクションを 1 つのセルフローとしてのセル多重化、1 つのフローを個別のコネクションに分けるセル分割、セルペイロード型付けと区別、セル損失優先度付けと軽減、セル受信とヘッダの精査、及びセルの汎用フロー制御を含む。これらの機能は、高伝送速度に対応できるようにハードウェアで実現するように設計される。ATM 層は、セルを伝送するために使用する物理媒体に依存しない。ATM 層の機能は、ATM セルヘッダ (図 2.14) のフィールドによって定義される。

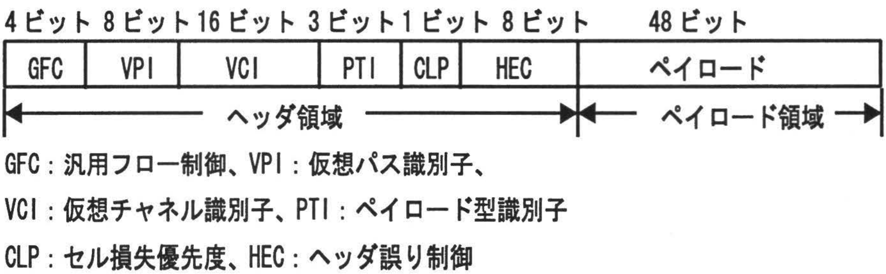


図 **2.14** ATM セル形式

■汎用フロー制御 **(generic flow control)**(GFC) フィールド:このフィールドは ATM ヘッダの内の 4 ビットである。既定値 0000 はセルが非制御であることを示す。非制御セルは、仮想回線が輻輳した時に他のセルを追い越さない。GFC フィールドのビットは、優先度のある輻輳制御を実現するために正しく設定することができる。例えば、この GFC フィールドのビットは映像よりも音声を優先する場合や、音声や映像が他のデータを追い越すことを示す場合に使用できる。GFC フィールドは、輻輳が発生した場合にネットワークへのデータ量を制限する UNI によるトラヒック制御にも使用される [Vetter 1995]。

■仮想パス識別子 **(virtual path identifier)**(VPI) と仮想チャネル識別子

(virtual channel identifier)(VCI) フィールド:VPI フィールドは8ビット、VCI フィールドは16ビットが ATM セルヘッダ内にとられる。これらの2つのフィールドは、ルーティングプロトコルがセルを転送する時のパスやチャネルを決定するために使用される。これらのフィールドは、パスにそって各ホップ毎に修正される。すなわち、セルが ATM 交換機に着くとその VPI と VCI の値は新しい仮想識別子を決定するために使用され、セルヘッダに設定され、セルを伝送するために出力リンクに渡される。その名前が示すように、VPI フィールドはネットワークのエンドポイント間の仮想パスを確立するために使用される。ATM では、2つのホスト間で複数のアプリケーションのセルが1つの仮想パスコネクション上にセル多重がなされる。これは、個々のアプリケーションのセルを区別し、仮想パス上で仮想リンクを確立する時にセルの VCI フィールドによって可能になる。

■**ペイロード型識別子 (payload-type identifier)**(PTI) フィールド:このフィールドは、ATM セルヘッダ内の3ビットである。これは、異なるサブチャネル上で転送できるユーザデータと制御データを区別するために使用される。

■**セル損失優先度 (cell loss priority)**(CLP) フィールド:このフィールドは、ATM セルヘッダ内の1ビットである。このビットは、ネットワークの輻輳が発生した時、セルが廃棄されたことを示す。例えば、テキストデータとは異なって音声データが再送することなくセル損失となる。この場合、アプリケーションは音声トラヒックに対してこの CLP フィールドを設定する。

■**ヘッダ誤り制御 (header error control)**(HEC) フィールド:このフィールドは、ATM セルヘッド内の8ビットである。このフィールドは、伝送誤りからヘッダフィールドを保護するために使用される。これは、ヘッダの検査合計を含む (ペイロードは含まない)。

ATM アダプテーション層 (ATM adaptation layer)

ATM プロトコルスウィートの物理層と ATM 層の機能は、アプリケーションに合ったものになっていない。ATM は各種のトラヒックタイプ、音声、映像、データをサポートする。これは、この AAL が特定の要求にしたがう各種のトラヒックタイプに応じてサービスを提供することによって実現している。これは、各種のトラヒックに応じたサービス品質の要求に応えるために 48 バ

サービスクラス	クラスA	クラスB	クラスC	クラスD
ビットレートタイプ	CBR	VBR	VBR	VBR
遅延特性	良	良	不定	不定
通信特性	不定	コネクション型	コネクション型	不定
AAL プロトコル	AAL1	AAL2 又は AAL5	AAL3/4 又は AAL5	AAL3/4

図 2.15 ATM アダプテーション層 (AAL) のサービスクラス

イトのセルとヘッダを構成している。アプリケーションの特性を反映するために、ITU では 4 つのサービスクラスを定義している (図 2.15)。

1. クラス A コネクション型サービスを要求し、遅延の少ない CBR トラヒックを持つアプリケーションはこのクラスに属する。映像や音声のアプリケーションが一般にこのクラスである。
2. クラス B コネクション型サービスを要求し、遅延の少ない VBR トラヒックを持つアプリケーションはこのクラスに属する。幾つかの映像や音声のアプリケーションがこのクラスである。
3. クラス C コネクション型サービスを要求するが、遅延については気にしない VBR トラヒックを持つアプリケーションがこのクラスに属する。このクラスの典型的なアプリケーションとしてコネクション型のファイル転送がある。
4. クラス D コネクション型サービスを要求しなく、かつ遅延についても気にしない VBR トラヒックを持つアプリケーションがこのクラスに属する。このクラスの典型的なアプリケーションとして LAN の相互接続や電子メールがある。

これらの 4 つのサービスクラスをサポートするために、まず ITU では AAL プロトコルの 4 つのタイプ、AAL1、AAL2、AAL3、AAL4 を勧告した。その後、AAL3 と AAL4 のプロトコルの差異は少ないため、AAL3/4 と呼ばれる 1 つのプロトコルにまとめられた。

さらに、AAL3/4 プロトコルの機構はコンピュータのデータトラヒックにとって複雑であるため、AAL5 と呼ばれる新たなプロトコルがコンピュータのデータトラヒックを処理するために AAL 層に追加された [Suzuki 1994]。AAL5

プロトコルは **SEAL (Simple and Efficient Adaptation Layer)**(簡易／効果アダプテーション層 (**Simple and Efficient Adaptation Layer**)) とも呼ばれる。

図 2.15 に示すように、クラス A のトラヒックは AAL1 プロトコルを使用し、クラス B のトラヒックは AAL2 プロトコルを使用し、クラス C と D のトラヒックは AAL3/4 又は AAL5 プロトコルを使用する。AAL3/4 と AAL5 プロトコルはクラス C と D のトラヒックに対応するため、この 2 つのプロトコルの違いを理解することは重要である。AAL3/4 は、各セルの誤り検出を行い、48 バイトのペイロードの 4 バイトを使って特別な誤り検査機構を持っている。AAL3/4 は ATM のセルを多重化することができる。一方、AAL5 は、従来の 5 バイトのヘッダ (セルのペイロードには何も影響しない) を使用するが、セルの多重化はサポートしない。

2.7.4　ATM ネットワーク

最も単純な構成として、ATM ネットワークは同軸若しくは光ファイバケーブルで接続された複数の ATM 交換機とこの ATM 交換機に接続されるホストコンピュータからなるメッシュ／スター型のアーキテクチャを持つ。図 2.16 は、3 台の ATM 交換機と 9 台のホストコンピュータからなる ATM ネットワークを示す。ホストコンピュータで発生したセルは接続する ATM 交換機を経由して他のホストコンピュータに届けられる。この単純な形態は ATM の LAN に対しても一般的に適用できる。ATM 交換機やホストコンピュータに加えて、広域の ATM ネットワークでは公衆網に相互接続する通信機器、例えばルータ、ゲートウェイ及びインタフェースを含む。

ATM 交換機は、幾つかの入出力ポートを持つ。各入力ポートは入力ポートコントローラを、各出力ポートは出力ポートコントローラを持つ。入力ポートコントローラは VCI テーブルとして参照されるテーブルを持ち、入力セルの VPI と VCI を出力 VCI と出力ポートアドレスに対応させる。入力セルが入力ポートコントローラによって交換機の交換部に渡される前に、セルの VCI は出力 VCI に書き換えられ、出力ポートアドレスがセルフルーティングのために付与される。各 ATM 交換機は、入力ポートコントローラのテーブルの更新を含む各種の交換機の管理機能を実行する交換機コントローラを持つ。さらに、各 ATM 交換機は幾つかの入力ポートから着たセルが同じ出力ポートに行く場合に一時的に蓄積するためのバッファを通常持つ。入力ポートと出力ポート毎に

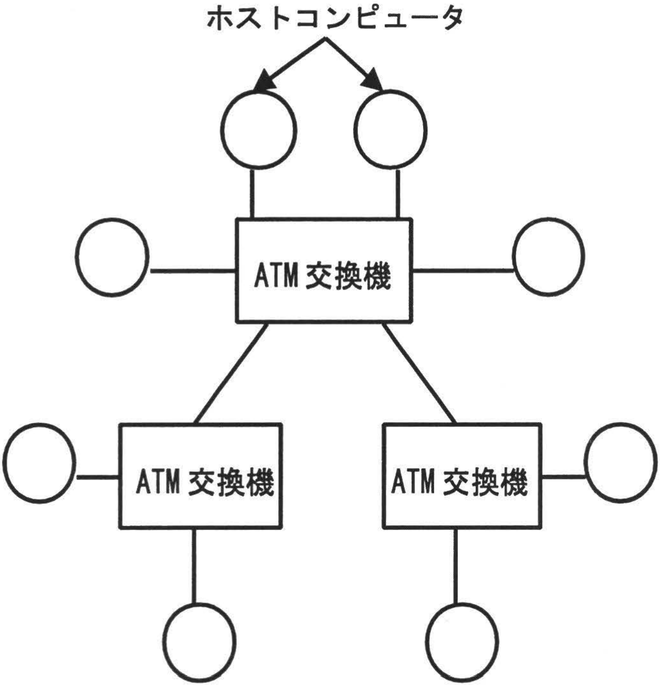


図 **2.16** ATM ネットワーク

別々のバッファを持つこともあれば、入出力ポート全体でバッファを持つこともある。

少ないポートを持つ ATM 交換機は、多くのポートを持つものに比べて安価で構築が容易である。通常、ATM の LAN では少ないポート数を持つ数少ない ATM 交換機で構成され、広域の ATM ネットワークでは多くのポートを持つ数多くの ATM 交換機によって構成される。

ATM ネットワークのホストコンピュータは、階層型の 8 バイト長の ISDN 電話番号方式 E。164 又は ATM フォーラム提案の 20 バイトのアドレス [ATM Forum 1993] によって ATM アドレスが割り当てられる。後者は、OSI ネットワークサービスアクセス点のアドレス形式に基づいている。

高位通信サービスのためのプロトコル支援 (例えば、TCP/IP、UDP/IP、Berkeley Software Distributor(BSD) ソケット及び RPC) を行うために ATM ネットワークは、既存のネットワークを相互接続するバックボーンネットワー

クとしても使用される。

2.7.5　課題と対応

ネットワーキング技術として、ATMは広帯域、規模拡張性、トラヒックの統合、統計多重及びネットワークの単純化など多くの有効な機能を備えている。これらの機能により、ATM技術は確かに将来の分散システムの設計における有効なものになっていくと考えられる。しかしながら、将来の分散システムのネットワーキング技術としてATMが成功するためには、幾つかの解決すべき課題がある。これらの課題は、ネットワーク設計者やユーザに新たな対応を投げかけている。このATM技術の分野で現在研究されている課題の中で重要なものについて幾つかその概要を次に説明する。

相互運用性

ATMがネットワーキング技術として成功するためには、既存のものを相互に接続することが必要である。このことは、既存の分散アプリケーションの多くがATMネットワーク上で動作し、ATMを受け入れるために重要である。このための注意すべきものとして、**ATMフォーラムでのLANエミュレーションオーバATM (LAN emulation over ATM)**(組織としてATM標準を開発し、定義した)[ATM Forum 1994]とIETF(インターネットエンジニアリングタスクフォース)[Chao et al. 1994, Laubach 1994, Brazdziunas 1994]での**IPオーバATM (IP over ATM)**の2つがある。これらについて以下に述べる。

LANエミュレーションオーバATM:LANエミュレーションオーバATM(**LANエミュレーション (LAN emulation)**と呼ばれる)は、既存のLANの分散アプリケーションがATMネットワークで動作するために関係する。これは、ATMネットワークと従来のLANの相互接続にも関係する。

多くの既存のLANは、共有メディアによる相互接続に基づいており、IEEE802のLANプロトコルであるイーサネット(802.3)、トークンバス(802.4)、トークンリング(802.5)(2.5.1節参照)を使用する。図2.9に示しているように、IEEE802モデルではOSI参照モデルのデータリンク層プロトコルは2つの層、伝送媒体をアクセス、共有、管理するための機構を定義する媒体アクセス制御(MAC)層と、異なるMAC層プロトコルとネットワーク層プロトコルを相互につなげるための共通インタフェースを定義する論理リンク制御(LLC)層の2

つに分割されている。LAN に接続する各ホストは、全体で唯一の MAC アドレスを持つ。MAC アドレスは、48 ビットからなり、単一のアドレス空間である。ホストは他のホストにデータを送る場合、その受信ホストの MAC アドレスを知っているか (2 つのホストとも同じ LAN の場合)、次のホップとなるルータの MAC アドレスを知らなければならない (ホストが異なる LAN にある場合)。

LAN エミュレーションの基本となる考え方は、AAL 層の上位でかつ LLC 層の下位に ATM-MAC 層として 1 つのプロトコル層を設計することである。この ATM-MAC 層によってサポートされなければならない主要な機能として次の 2 つがある。

1. ブロードキャスト通信機能を実現するために、LAN の物理、ブロードキャスト、共有媒体のエミュレーション
2. 2 地点間通信機能を実現するために、MAC アドレスから ATM アドレスの解決

ATM は、LAN エミュレーションクライアント (LEC) として参照される ATM ネットワークに直接接続されるホスト間の ATM マルチキャスト仮想コネクションを確立することによって、従来の LAN の物理的ブロードキャストによる共有媒体をエミュレーションする。このマルチキャストコネクションは、ATM LAN セグメントにおけるブロードキャストのチャネルである。いずれの LEC もマルチキャスト仮想コネクションを使って ATM LAN セグメント上の全てのホストにブロードキャストできる。

2 地点間通信については、アドレス解決プロトコルが 48 ビットの MAC アドレスから ATM アドレスに変換するために必要となる。MAC アドレスから ATM アドレスへの変換は、ブロードキャストに基づく方法か若しくはサーバに基づく方法で行われる。ブロードキャストに基づく方法は、交換機のブロードキャスト能力に関係する。一方、サーバに基づく方法では LAN エミュレーションサーバ (LES) が全体で知られている仮想チャネル上に置かれ、問い合わせと応答メッセージが LES と LEC 間で交信される。全ての MAC から ATM へのアドレス解決の要求は、LES に送られ、予め決められている仮想チャネルを使用して要求元に応答する。

受信ホストの ATM アドレスが得られると、2 地点間の ATM 仮想コネクションが ATM シグナリングプロトコルを使って送信側ホストと受信側ホスト間に確立される。アドレス解決の結果や確立されたコネクションの VCI は、受信側

ホストとの通信が今後もあるものとして送信側ホストのテーブルに一時的に保持される。この機構はATM-MAC層内で行われ、ホスト内のLLC層プロトコルや上位層プロトコルに対しては透過である。LANエミュレーションの詳細については[Newman 1994]を参照されたい。

IPオーバATM:IPオーバATMは、IPスウィートを意識して設計された既存の分散アプリケーションがATMネットワーク上で動作するためのものである。IPオーバATMを実現するために考えなければならない点は下記の2つである。

1. LANエミュレーション機能の上でのIPオーバATMの実現
2. ATM交換機のみで構成される純粋なATMネットワーク上でのIPの実現

LANエミュレーションがATM-MAC層をAAL層の上位でかつLLC層の下位に定義したように、LANエミュレーション上のIPの実現はIEEE802のLANの上にIPを実現することと同じである。しかしながら、IPが純粋なATMネットワーク上で実現される場合には、プロトコルスタックを簡単にでき、ATM上で直接動作させることができる。ATM上で直接IP動作させるためには、IPアドレスからATMアドレスに変換するためのアドレス解決プロトコルが必要である。この場合、サーバに基づく方法(LANエミュレーションで記述)が再びこのために使用される。IPオーバATMでは、アドレス解決サーバはIP-ATM-ARPサーバと呼ばれ、IPアドレスとATMアドレスの対応情報を含むテーブルを管理する。

新しいホストがATMネットワークに追加されると、まず登録処理が実行され、それのATMアドレスが得られる。そしてIP-ATM-ARPサーバにメッセージを送り、テーブルにアドレスリゾルーションの情報を追加する。このメッセージは、新しく追加したホストのIPとATMのアドレスを含む。サーバはテーブルを更新し、そのホストに新たなVCIを割り当て、ホストに通知する。ホストはサーバから受信したメッセージと他のホストからのものを識別するためにこのVCIを使用する。このような処理は、ホストの移動に伴っても行われる。IPオーバATMの詳細については[Chao et al. 1994]を参照されたい。

帯域管理 (bandwidth management)

ATMネットワークは、同期性、非同期性及び等時性の混在したトラヒックを

伝送するための手段である。要求されるサービス品質を達成するために、ユーザはアプリケーションに必要となる帯域を指定することが要求される。コネクション上で解決しなければならない2つの重要な問題は、ユーザはアプリケーションに必要な帯域をどのように算出し、指定できるか、それとネットワークとしてアプリケーションの要求を満足させるために可能な帯域からアプリケーションにどのように割り当てるかである。これに関して研究者からの提案を次に説明する [Turner 1992, Vetter 1995]。

1. **最大レート割り当て方式 (peak-rate allocation method)**:この方式では、ユーザはアプリケーションの最大トラヒックレートのみを指定する。ユーザの要求に基づいて、ネットワークは全てのネットワークのリンクにおいて仮想チャネルのレートの総和がそのリンクの最大セルレートを越えない範囲でアプリケーションに対して仮想チャネルを割り当てる。アプリケーションのトラヒックがこの最大レートを越えた場合には、セルは単に廃棄される。

2. **最小スループット割り当て方式 (minimum-throughput allocation method)**:この方式では、ユーザはアプリケーションの必要な最小スループットを指定し、ネットワークは最善努力によりアプリケーションに対して指定されたスループットを保証する。

3. **バース性トラヒック指定方式 (bursty-traffic specification method)**:この方式では、ユーザはアプリケーションの最大と平均トラヒックレートに加えて最大バーストサイズを指定する。これらのパラメータは、要求に合うように可能な帯域を割り当てためにネットワークを適切に構成するために使用される。

これらのどれも十分であるとは言えない。例えば、Vetter[1995] によると、最初の方式は性能を保証し、実現は容易であるが、バース性トラヒックに対しては帯域の利用効率が悪い。2番目の方式は効率的であるが、保証される性能について弱い。3番目の方式は新しい仮想チャネルを使用している仮想チャネルに加える場合かなりの計算が必要となり、オーバヘッドが生じる。さらに、全ての方式において2つの問題を持っている。1つはエンドツーエンドのプロトコルでは通常複数のセルからなるデータ単位で動作しているため、信頼性のあるサービスを要求するアプリケーションにとって1つのセルの損失や廃棄に

よって全体の再送を行わなければならなく、その結果スループットの低下を招く [Vetter 1995]。この問題は、多くのデータ量の再送を引き起こすことから広域の ATM ネットワークにおいて顕著になる。これは悲観的な性能を導く。2番目の問題は、3つの方式ともマルチキャスト仮想回線を適切に処理できない点である [Vetter 1995]。したがって、このような問題を解決できる新しい帯域管理機構が望まれる。

遅延と帯域のトレードオフ

Kleinrock[1992] は、高速な広域の ATM ネットワークにおける遅延と帯域のトレードオフはこれらのネットワークの設計者に対して新たな挑戦を与えていると指摘している。これは広域での伝搬遅延又は光の速度が ATM セルを伝送する時間よりも大きいことに起因している。例えば、一方向に1ビットがアメリカを横断するのに約 15 ms かかる。1 Gbps では、この時間は1つの ATM セルをリンクに伝送するのに必要となる時間の 35,000 倍以上である [Vetter 1995]。このような遅延と帯域のトレードオフは、次のような2つの主な問題を提起している。同じ問題が [Tanenbaum 1995] に記述されている。

1. アメルカ全体の WAN を考えよう。WAN における2つのサイト間の往復の伝搬遅延は、約 30 ms と見なせる。この2つのサイトが 622 Mbps の ATM ネットワークでつながっているとする。622 Mbps では、1メガビットのファイルをネットワークに送出するのに約 1.6 ms(1/622) である。2つのサイト間の往復伝搬遅延が 30 ms であるため、受信側からの応答は受信側で約 31.6 ms 後に受ける。したがって、この例では全体の 31.6 ms の 95 %、30 ms がリンクとしてはアイドル状態となる。このような状況は、より高速で短メッセージであれば顕著になる。より速い速度では、効率良く使用できる可能な仮想回線の帯域は0に近づく。したがって、高速な広域の ATM ネットワークでは遅延の問題に対応できる新しいプロトコルとシステムアーキテクチャが必要である。

2. 伝搬遅延は ATM セルの伝送時間よりも数倍も大きいため、送信側は受信側に最初のビットが届く前に数千個のセルを送信できる。受信側がバッファを十分に持っていなければセルの大半は失ってしまい、再送を行うことになる。これは悲観的な性能を意味する。この問題に対してフロー制御として従来のスライド型のウィンドウプロトコルの使用は、良い結果をもたらさない。これは、送信側がメガビットのデータ毎に応答を待つとすると既に述べた遅延の問題のように、仮想回線の 95 %がアイドルとなる。この問題を解決するためには、フロー制御、バッファリング及び輻輳制御に関して検討が必要である。

これらについて幾つかの検討がなされている [Hong and Suda 1991, Trajkovic and Golestani 1992, Eckberg 1992, Yazid and Mouftah 1992]。

2.8 まとめ

分散システムは、構成するノード間のデータや制御情報の通信のためにコンピュータネットワークの上に構築される。コンピュータネットワークは、ネットワーク上の 2 つのノードのプロセス間でのデータ交信のために通信回線やソフトウェアプロトコルによってノードを接続する通信システムである。

ノードの地理的な配置、データ速度、誤り率及び通信費用などの特性に基づいて、ネットワークは大きく 2 つのタイプ、LAN と WAN に分類される。この LAN と WAN の両者の特性を持つネットワークとして MAN がある。

LAN を構成するネットワーク形状として 2 つあり、多重アクセスのバスとリングである。これらの形状のネットワークでは、単一のチャネルがネットワーク全体のサイトによって共有される。したがって、メディアアクセス制御プロトコルが全てのサイトによって共有されるチャネルへのアクセスを制御するために必要となる。多くのメディアアクセス制御プロトコルにおいて、CSMA/CD プロトコルが多重アクセスのバスネットワークで最も共通的に使用さ、トークンリングやスロッテッドリングプロトコルはリングネットワークで使用される。

コンピュータの広域ネットワークは、遠く離れたコンピュータを接続するものである。パケット交換機 (PSE) と呼ばれる特殊な装置がコンピュータを通信チャネルに接続するために使用される。この PSE は、ネットワークにデータを転送するための交換及びルーティング処理を行う。共通的な 2 つの交換技術として、回線交換とパケット交換がある。

WAN においてパケットを転送するために使用されるパスの選択は、使用されるルーティング方式によって決まる。あるサイトから別のサイトにパケットを転送するためのパスは、固定かネットワークの状態によって動的に変化する。これは、使用する静的若しくは動的なルーティング方式に依存する。さらに、全体のパスが送信元で決定される場合と、各ホップのパスが各サイトによって決定される場合がある。また、動的なルーティング方式では、ルーティングテーブルは個別方式、集中方式又は分散方式によって更新される。

コンピュータネットワークは、階層化プロトコルの概念に基づいて実現される。OSI モデルは WAN における階層化プロトコルの標準であり、IEEE802 LAN モデルは LAN の標準である。OSI モデルの 7 層は、物理層、データリン

ク層、ネットワーク層、トランスポート層、セション層、プレゼンテーション層及びアプリケーション層である。一方、IEEE802 LAN モデルの最下位3層は物理層、メディアアクセス制御層及び論理リンク制御層である。この LAN モデルは、上位層を定義していない。ネットワークシステムの利用可能なプロトコルスウィートとして、インターネットプロトコル (IP) スウィートが最も良く使用されている。

ネットワークシステムのために設計される通信プロトコルは、ネットワークシステムに比べて分散システムに対する要求が特殊であるため分散システムに通常適用し難い。幾つかの通信プロトコルは、分散システムでの高いスループットや速い応答さらに分散システムの特殊な要求を満たすように設計されている。このような特徴を持つ2つのプロトコルとして VMTP と FLIP を本章で説明した。

複数のネットワークを相互接続して1つのネットワークにすることをインターネットワーキングと呼ぶ。この1つのネットワークはインターネットワークと呼ばれる。インターネットワーキングの目標は、個別の物理ネットワークの詳細を隠蔽し、単一の統合化されたインターネットワーキングの機能を提供することである。このための装置としてブリッジ、ルータ、ブルータ、ゲートウェイなどがある。インターネットは、インターネットワークの最も良い事例である。

ATM は、高速なコネクション型の交換／多重化技術であり、音声、映像、データなどを含む異なるタイプのトラヒックを同時に転送するために短い固定長のパケット (セルと呼ばれる) を使用する。ATM 技術は、多くの有効な機能から将来のコンピュータネットワーキング技術として期待されている。

2.9　演習問題

2.1 LAN と WAN の主な相違は何か？

2.2 メディアアクセス制御プロトコルは何故必要か？良いメディアアクセス制御プロトコルはどのような機能を持つべきか？

2.3 メディアアクセス制御として CSMA/CD 方式は、共有媒体のランダムアクセス、衝突検出、衝突検出後の再送を行う。衝突が起きないように (衝突を全体として避ける) するためにメディアアクセス制御としてのランダムアクセス方式を工夫できるか？ もしできない場合はその理由を述べよ。できる場合にはその方式とその利点と欠点を述べよ。

2.4 イーサネットに関する次の質問に応えよ。

(a) パケットを転送するためのサイトの共有チャネルの獲得方法

(b) 衝突検出方法

(c) 検出後の衝突回避方法

(d) 繰り返される衝突の最小確率

2.5 イーサネットにおいて、衝突間隔は媒体の端から別の端に信号が伝搬し、戻ってくるのに必要な時間であり、また再送スロット時間は衝突間隔よりも少し大きく設定される。全てのサイトが共有媒体が少なくとも再送スロット時間と等しい時間使われていないことを確認してパケットを送信すれば衝突が起きないことを証明せよ。

2.6 トークンリングとスロッテッドリングプロトコルは、ネットワークとして各サイトを論理的にリング形状に構成する。サイトの障害検出方法及びサイトの障害が検出された時の論理リングの再構成法を説明せよ。

2.7 メディアアクセス制御のトークンリング方式におけるトークン紛失の検出方法を述べよ。

2.8 メディアアクセス制御の優先度付きトークンリング方式において高優先度のサイトが何時も送信するデータを持っている時低優先度のサイトにも送信権が渡るための方法を述べよ。

2.9 高速 LAN おいて、ここでの高速の意味は平均パケット伝送時間が伝搬遅延よりも小さくなるような十分速いデータ伝送速度とする。バス型 LAN は 6000m で平均パケット長は 1400 ビット、データ伝送速度は 50 Mbps とする。この LAN の伝搬速度が 200,000 km／s の時、この LAN は高速 LAN と言えるか？

2.10 CSMA/CD の性能は、$a = (\tau\ W)/B$ の値が増加すると著しく低下することが示されている。ここで、τ はネットワークのエンドツーエンドの伝搬遅延、W はチャネルの帯域、B はパケットのビット長である。また、ケーブル (500m 毎にリピータを含めて) の 1 km 当たりの τ は 10 マイクロ秒である。実験結果として、$a = 0.1$ の時 CSMA/CD はチャネル容量として 0.65 以上を提供することが示されている。問題 2.9 で定義される高速 LAN において、CSMA/CD 方式は高速 LAN として適切であるかを示せ。

2.11 インターネットワーキングとは何か？ インターネットワーキングにおける主な問題は何か？ 下記の用語の相違がわかるように説明せよ。

(a) ブリッジ

(b) ルータ

(c) ゲートウェイ

2.12 回線交換においてデータを転送する場合に次のメッセージシーケンスを仮定する。

(a) コネクト要求 (送信側から受信側)

(b) コネクション確認通知 (受信側から送信側)

(c) データ送信 (送信側から受信側へのデータ送信)

(d) データ確認通知 (受信側から送信側)

(e) コネクト切断要求 (送信側から受信側)

(f) コネクション切断確認通知 (受信側から送信側)

上記の各メッセージの転送時間は t で、ただしデータ送信の転送時間は t + ds で d は定数、s はデータのバイト数とする。ここで、同じネットワークにおいて、t は 100 バイトのパケットを転送するに必要な時間とし、パケット交換では n パケットを受ける毎に受信側から送信側に応答が返されるものとする。同じデータ伝送速度において回線交換とパケット交換を s の値によって比較せよ。

2.13 コンピュータネットワークで使用される 3 種類のルーティング方式を述べ、その方式の利点と欠点を示せ。

2.14 ネットワークシステムは動的なルーティング方式を使用し、非最小パス選択方式とする。このシステムにおいて、パケットがネットワークを転送されたが宛先に届かない。このような状態を避ける方法を述べよ。

2.15 ネットワークシステムにおいて何故通信プロトコルは必要か？ 通信プロトコルの仕様や設計において階層化手法を使用する主な理由は何か？

2.16 OSI モデルのアーキテクチャの図を示し、各層の機能の概要を説明せよ。

2.17 図 2.8 で各層でメッセージに対してヘッダが付与させることを説明した。このように各層でのヘッダ付与に代わって、ネットワークに送出する前に全体として 1 つのヘッダを付与する方法を考える。この方法は何故できないか？

2.18 多くのコンピュータネットワークは、OSI モデルとは異なって少ない階層を使用している。この理由は何か？ どのような問題でこのようになっているか？

2.19 コネクション型通信プロトコルとコネクションレス型通信プロトコルの主な相違は何か？ 次の情報タイプの伝送に適するのはどちらのプロトコルかを示せ。

(a) 音声

(b) 映像

(c) バース性のデータ

2.20 メッセージのパケットは、送信した時の順番とは違った順番で宛先に届くことがある。次のネットワークにおいてこの事が正しいかを理由とともに示せ。

(a) イーサネット LAN

(b)WAN

(c)ATM LAN

(d)ATM WAN

2.21 何故 OSI モデルは LAN 環境では適応しないと考えられるか？ LAN に最適な通信プロトコルモデルのアーキテクチャを示せ。そのアーキテクチャの各層の機能の概要を述べよ。

2.22 一般に従来のネットワークシステムの通信プロトコルは何故分散システムに適さないか？

2.23 次の点について VMTP プロトコルでの方法を説明せよ。

(a) メッセージ損失の処理

(b) グループ通信

(c) フロー制御

(d) 透過通信

2.24 次の点について FLIP プロトコルでの方法を説明せよ。

(a) 透過通信

(b) グループ通信

(c) セキュア通信

(d) 容易なネットワーク管理

2.25 ATM 技術の主要な機能は何か？ これらの機能は将来の分散システムにおいてどのようなインパクトを与えるか？

2.26 ATM プロトコル参照モデルの各層の機能を説明せよ。

2.27 次の機能はどのように実現されるか説明せよ。

(a)LAN エミュレーションオーバ ATM

(b)IP オーバ ATM

2.28 アプリケーションの要求を満たすように可能な帯域を最適に割り当てるための ATM ネットワークにおける 3 種類の方式を示せ。またこれらの方式の相対的な特長と制限を述べよ。

2.29 高速で広域の ATM ネットワークにおける遅延と帯域のトレードオフに関して起きる問題を具体例で説明せよ。

参考文献

[Aveysundara and Kamal 1991] Abeysundara, B. W., and Kamal, A. E., "High-Speed Local Area Networks and Their Performance: A Suravey," *ACM Computing Suraveys*, Vol. 23, No. 2, pp.221-264 (1991).

[ATM Forum 1993] *ATM Use-Network Interface Specification Version 3.0*, Prentice-Hall, Englewood Cliffs, NJ (1993).

[ATM Forum 1994] "LAN Emulation Over ATM," Draft Specification—Revision 5 (ATM FORUM 94-0035R5), LAN Emulation Sub-Working Group of the ATM Forum Technical Committee (August 1994).

[Black 1993] Black, U., *Computer Networks: Protocols, Standards, and Interface*, 2nd ed., Prentice-Hall, Englewood Cliffs, NJ (1993).

[Black 1995a] Black, U., *ATM: Roundation for Broadband Networks*, Prentice-Hall, Englewood Cliffs, NJ (1995).

[Black 1995b] Black, U., *TCP/IP and Related Protocols*, 2nd ed., IEEE Computer Society Press, Los Alamitos, CA (1995).

[Brazdziunas 1994] Brazdziunas, C., "Iping Support for ATM Services," Internet RFC No. 1680 (1994).

[Chao et al. 1994] Chao, H. J., Ghosal, D., Saha, D., and Tripathi, S. K., "IP on ATM Local Area Networks," *IEEE Communications Magazine*, pp.52-59, New York, NY (August 1994).

[Cheriton 1986] Cheriton, D. R., "VMTP: A Transport Protocol for the Next Generation of Communication Systems," In: *Proceedings of the SIGCOMM'86*, pp.406-415 (August 1986).

[Cheriton 1988] Cheriton, D. R., "The V Distributed System," *Communications of the ACM*, Vol. 31, No. 3, pp.314-333, ACM, New York, NY (1988)

[Cheriton and Williamson 1989] Cheriton, D. R., and Williamson, C.L., "VMTP As the Transport Layer for High-Performance Distributed Systems," *IEEE Communications Magazine*, Vol. 27, No. 6, pp.37-44, New York, NY (1989).

[Comer 1995] Comer, D. E., *Internetworking with TCP/IP: Volume 1—Principles, Protocols, and Architectures*, 3rd ed., Prentice-Hall, Englewood Cliffs, NJ (1995).

[Comer and Stevens 1993] Comer, D. E., and Stevens, D. L., *Internetworking with TCP/IP: Volume3—Client-Server Programming and Applications: BSD Socket Version*, Prentice-Hall, Englewood Cliffs, NJ (1993).

[Comer and Stevens 1994] Comer, D. E., and Stevens, D. L., *Internetworking with TCP/IP: Volume 2—Design. Implementation. And Internals*, 2nd ed., Prentice-Hall, Englewood Cliffs, NJ (1994).

[Datapro 1990] Datapro, "An Overview of Simple Network Management Protocol," Datapro Network Management. NM40-300-201 (February 1990).

[Datapro 1993] Open Software Foundation (OSF) Distributed Management Environment (DME), A Datapro Report, Datapro Network Management, NM40-684-07(April 1993).

[DePrycker 1993] DePrycker, M., *Asynchronous Transfer Mode: Solution for Broadband ISDN*, 2nd ed., Ellis Horwood (1993).

[DePrycker et al.1993] DePrycker, M., Peschi, R., and Landegem, T., "B-ISDN and the OSI Protocol Reference Model," *IEEE Network*, Vol. 7, No. 2, pp.10-18, New York, NY (March 1993).

[Dutton and Lenhard 1995] Dutton, Jr., H., and Lengard, P., *High-Speed Networking Technology: An Introductory Survey*, Prentice-Hall, Englewood Cliffs, NJ(1995).

[Eckberg 1992] Eckberg, A., "B-ISDN/ATM Traffic and Congestion Control," *IEEE Network*, Vol. 6, No. 5, pp.28-37, New York, NY (1992).

[Finlayson et al. 1984] Finlayson, R., Mann, T., Mogul, J., and Theimer, M., "A Reverase Address Resolution Protocol," RFC No. 903 (June 1984).

[Fischer et al. 1994] Fischer, W., Wallmeier, E., Worster, T., Davis, S. P., and Hayter, A., "Data Communications Using ATM: Architectures, Protocols, and Resource Management," *IEEE Communicaions Magazine*, pp.24-33, New York, NY (August 1994).

[Furht and Milenkovic 1995] Furht, B., and Milenkovic, M., *Guided Tour of Multimedia Systems and Applications*, IEEE Computer Society Press, Los Alamitos, CA (1995).

[Haendel et al. 1994] Haendel, R., Huber, M. N., and Schroeder, S., *ATM Networks Concepts, Protocols, Applications*, Addison-Wesley, Reading, MA(1994).

[Helgert 1991] Helgert, H. J., *Integrated Services Digital Networks: Architectures, Protocols, Standards*, Addison-Wesley, Reading, MA (1991).

[Hon and Suda 1991] Hong, D., and Suda, T., "Congestion Control and Prevention in ATM Networks," *IEEE Network*, Vol. 5, No. 4, pp.10-16, New York, NY (1991).

[Hughes 1994] Hughes, K., "Entering the World-Wide Web: A Guide to Cyberspace,"
ftp:taurus.cs.nps.navy.mil:/pub/mbmy/world-wide-web-buide.ps.z (1994).

[IEEE 1985a] *Carrier Sense Multiple Access with Collision Detect(CSMA/CD) Access Method and Physical Layer Specifications*, ANSI/IEEE 802.3 (ISO/DIS 8802/3), IEEE, New York (1985).

[IEEE 1985b] *Token-Passing Bus Access Method and Physical Layer Specifications*, ANSI/IEEE 802.4 (ISO/DIS 8802/4), IEEE, New York (1985).

[IEEE 1985c] *Token Ring Access Method and Physical Layer Specifications*, ANSI/IEEE 802.5 (ISO/DIS 8802/5), IEEE, New York (1985).

[IEEE 1990] *IEEE Standard 802: Overview and Architecture, American National Standard ANSI/IEEE 802*, IEEE Computer Society Press, Los Alamitos, CA (1990).

[IT 1990] Information Technology—Open Systems Interconnection—Management Information Protocol Specification—Common Management Information Protocol, ISO/IEC 9596-1, ISO/IEC JTC1/SC21 N5303 (November 1990).

[Janet 1993] Janet, E. L., "Selecting a Network Management Protocol, Functional Superiority vs. Popular Appeal," *Telephony* (November 1993).

[Kaashoek et al. 1993] Kaashoek, M. F., Van Renesse, R., Staveren, H., and Tanenbaum, A. S., *Computer Systems*, Vol. 11, No. 1, pp.73-106 (1993). ©ACM, Inc., 1993.

[Kawarasaki and Jabbari 1991] Kawarasaki, M., and Jabbari, B., "B-ISDN Architecture and Protocol," *IEEE Journal of Selected Areas on Communications*, Vol. SAC-9, No. 9, pp.1405-1415, New York, NY (1991).

[Kim and Wang 1995] Kim, B. G., and Wang, P., "ATM Network: Goals and Challenges," *Communications of the ACM*, Vol. 38, No. 2, pp.39-44 (1995).

[**King and Mitrani 1987**] King, P. J. B., and Mitrani, I., "Modeling a Slotted Ring Local Area Network," *IEEE Transactions on Computers*, Vol. C-36, No. 5, pp.554-561, Piscataway, NJ (1987).

[**Kleinrock 1992**] Kleinrock, L., "The Latency/Bandwidth Tradeoff in Gigabit Networks," *IEEE Communications Magazine*, Vol. 30, No. 4, pp.36-40, New York, NY(1992).

[**Krol 1994**] Krol, E., *The Whole Internet: User's Guide and Catalog*, 2nd ed., O'Reilly, Sebastopol, CA (1994).

[**Kung 1992**] Kung, H. T., "Gigabit Local Area Networks: A Systems Perspective," *IEEE Communications Magazine*, Vol. 30, No. 4, pp.79-89, New York, NY (1992).

[**Larmouth 1993**] Larmouth, J., *Understanding OSI*, Prentice-Hall, London, UK (1993).

[**Laubach 1994**] Laubach, M., "Classical IP and ARP over ATM," RFC No. 1577 (January 1994).

[**Leinwand and Conroy 1996**] Leinwand, A., and Conroy, K. f., *Network Management: A Practical Perspective*, 2nd ed., Addison-Wesley, Reading, MA (1996).

[**Macedonia and Brutzman 1994**] Macedonia, M. R., and Brutzman, D. P., "*Mbone Provides Audio and Video Across the Internet*," IEEE Computer, pp.30-36 (April 1994).

[**Malamud 1992**] Malamud, C., STACKS: *Interoperability in Today's Computer Networks*, Prentice-Hall, Englewood Cliffs, NJ (1992).

[**Malamud 1993**] Malamud, C., *Exploring the Internet: A Technical Travelogue*, Prentice-Hall, Englewood Cliffs, NJ (1993).

[**Martin 1993**] Martin, J. L., "Travels with Gopher," *IEEE Computer*, Vol 26., No. 5,pp.84-87 (1993).

[**Miller 1995**] Miller, M. A., *Inter Networking: A Guide to Network Communications LAN to LAN; LAN to WAN*, 2nd ed., M&T Books, New York, NY (1995).

[**Mosaic 1994**] "What's New with NCSA Mosaic," http:www.ncsa.uiuc.edu/SDG/Software/Mosaic/Docs/Watsnew.html (June 1994).

[Mullender et al. 1990] Mullender, S. J., Van Rossum, G., Tanenbaum, A. S., Van Renesse, R., and Van Staverene, H., "Amoeba: A Distributed Operating System for the 1990s," *IEEE Computer*, Vol. 23., No. 5, pp.44-53(1990).

[Nejmeh 1994] Nejmeh, B. A., "Internet: A Strategic Tool for the Software Enterprise," *Communications of the ACM*, Vol. 37, No. 11, pp.23–27(1994). ©ACM, Inc., 1994.

[Newman 1994] Newman, P., "ATM Local Area Networks," *IEEE Communications Magazine*, Vol. 32, No. 3, pp. 86–98 (1994).

[Omidyar and Aldridge 1993] Omidyar, C., and Aldridge, A., "Introduction to SDH/SONET," *IEEE Communications Magazine*, Vol. 31, No. 9, pp.30-33 (1993).

[Partridge 1994] Prtridge, C., *Gigabit Networking*, Addison-Wesley, Reading, MA (1994).

[Perlman 1992] Perlman, R., *Interconnections: Bridges and Routers*, Addison-Wesley, Reading MA (1992).

[Pllummer 1982] Plummer, D. C., "An Ethernet Address Resolution Protocol," RFC No. 826 (November 1982).

[Postel 1980] Postel, J., "User Datagram Protocol," RFC No. 768, USC Information Sciences Institute (August 1980).

[Postel 1981a] Postel, J., "Internet Protocol: DARPA Internet Program Protocol Specification," RFC No. 791 (September 1981).

[Postel 1981b] Postel, J., "Internet Control Message Protocol," RFC No. 792 (September 1981).

[postel 1981c] Postel, J., "Transmission Control Protocol: DARPA Internet Program Progocol Specification." RFC No.793 (September 1981).

[Press 1994] Press, L., "Commercialization of the Internet," *Communications of the ACM*, Vol. 37, No. 11, pp.17-21 (1994).

[Ramos et al. 1996] Ramos, E., Schroeder, A., and Beheler, A., *Computer Networking Concepts*, Prentice-Hall, Englewood Cliffs, NJ (1996).

[Rooholamini 1995] Rooholamini, R., "ATM-Based Multimedia Servers," *IEEE Multimedia*, pp.39-52 (Spring 1995).

[Santifaller 1994] Santifaller, M., *TCP/IP and ONC/NFS, Internetworking in a UNIX Environment*, 2nd ed., Addison-Wesley, Reading, MA (1994).

[Shevenell 1994] Shevenell, M., "NMPv2 Needs Reworking to Emerge as a Viable Net Management Platform, "*Network World* (March 7, 1994).

[Smythe 1995] Smythe, C., *Internetworking: Designing the Right Architectures*, Addison-Wesley, Reading, MA (1995).

[Stallings 1992a] Stallings, W. (Ed.), *Advances in ISDN and Broadband ISDN*, IEEE Computer Society Press, Los Alamitos, CA (1992).

[Stallings 1992b] Stallings, W., *Computer Communications: Architectures, Protocols and Standards*, 3rd ed., IEEE Computer Society Press, Los Alamitos, CA (1992).

[Stallings 1993a] Stallings, w. (Ed.), *Advances in Local and Metropolitan Area Networks*, IEEE Computer Society Press, Los Alamitos, CA (1993).

[Stallings 1993b] Stallings, W. (Ed.), *Network Management*, IEEE Computer Society Press, Los Alamitos, CA (1993).

[Stallings 1993c] Stallings, W., *Networking Standards: A Guide to OSI, ISDN, LAN, and MAN Standards*, Addison-Wesley, Reading, MA (1993).

[Stallings 1995] Stallings, W., *ISDN and Broadband ISDN with Frame Relay and ATM*, 3rd ed., Prentice-Hall, Englewood Cliffs, NJ (1995).

[Suzuki 1994] Suzuki, T., "ATM Adaptation Layer Protocol," *IEEE Communications Magazine*, Vol. 32, No. 4, pp.80-83 (1994).

[Tanenbaum 1988] Tanenbaum, A. S., *Computer Networks*, 2nd ed., Prentice-Hall, Englewood Cliffs, NJ (1988).

[Tanenbaum 1995] Tanenbaum, A. S., *Distributed Operating Systems*, Prentice-Hall, Englewood Cliffs, NJ (1995).

[Tittel and James 1996] Tittel, E., and James, S., *ISDN Networking Essentials*, Academic Press, San Diego, CA (1996).

[Trajkovic and Golestani 1992] Trafkovic, L., and Golestani, S. J., "Congestion Control for Multimedia Services," *IEEE Network*, Vol. 6, No. 5, pp.20-26 (1992).

[Turner 1992] Turner, J., "Managing Bandwidth in ATM Networks with Burstyy Traffic," *IEEE Network*, Vol. 6, No. 5, pp.50-58 (1992).

[Umar 1993] Umar, A., *Distributed Computing: A Practical Approach*, Prentice-Hall, Englewood Cliffs, NJ (1993).

[Verma 1990] Verma, P. K., *ISDN Systems: Architecture, Technology & Applications*, Prentice-Hall, Englewood Cliffs, NJ (1990).

[Vetter 1995] Vetter, R. J., "ATM Concepts: Architectures, and Protocols," *Communications of the ACM*, Vol.38, No.2, pp.31-38 (1995).

[Vetter and Du 1993] Vetter, R. J., and Du, D. H. C., "Distributed Computing with High-Speed Optical Networks," *IEEE Computer*, Vol. 26, No. 2, pp.8-18 (1993).

[Vetter et al. 1994] Vetter, R. J., Spell, C., and Ward, C., "Mosaic and the World-wide Web," *IEEE Computer*, pp.49-56 (October 1994).

[Vickers and Suda 1994] Vickers, B. J., and Suda, T., "Connectionless Service for Public ATM Networks," *IEEE Communications Magazine*, pp.34-42 (August 1994).

[Voruganti 1994] Boruganti, R. R., "A global Network Management Framework for the '90s," *IEEE Communications Magazine*, pp.74-83 (August 1994).

[Wilkes and Wheeler 1979] Wilkes, M. V., and Wheeler, D. J., "The Cambridge Digital Communication Ring," In: *Proceedings of the Local Area Communications Network Symposium*, Boston, pp.47-61 (May 1979).

[Wittie 1991] Wittie, L. D., "Computer Networks and Distributed Systems," *IEEE Computer*, Vol. 24, No. 9, pp.67-75 (1991).

[WWW 1994a] "World-Wide Web Growth," ftp:nic.merit.edu (1994).

[WWW 1994b] "The WWW Virtual Library," http:info.cern.ch/hypertext/DataSources/bySubject/Overview.html (June 1994).

[Yazid and Mouftah 1992] Yazid, S., and mouftah, H. T., "Congestion-Control Methods for B-ISDN," *IEEE Communications Magazine*, Vol.30, No.7, pp.42-47 (1992).

インターネット参考文献

コンピュータネットワーキングに関する参考文献は、以下のところで見つけることができる。
ftp:ftp.cs.umanitoba.ca/pub/bibliographies/Distributed/network.html
ftp:ftp.cs.umanitoba.ca/pub/bibliographies/Distributed/CCR.html

ネットワーク相互接続における通信とルーティングに関する参考文献は、以下のところで見つけることができる。
ftp:ftp.cs.umanitoba.ca/pub/bibliographies/Parallel/par.comm.html

インターネットに関する参考文献は、以下のところで見つけることができる。
ftp:ftp.cs.umanitoba.ca/pub/bibliographies/Misc/internet.html

ギガビット又は高速ネットワークに関する参考文献は、以下のところで見つけることができる。
ftp:ftp.cs.umanitoba.ca/pub/bibliographies/Distributed/gigabit.html

ATM ネットワークに関する参考文献は、以下のところで見つけることができる。
ftp:ftp.cs.umanitoba.ca/pub/bibliographies/Distributed/ATM.html

第3章

メッセージパッシング

3.1　概要

プロセス (process) は実行できるプログラムである。分散システムにおける2つのコンピュータがお互いに通信していると言った場合、各コンピュータ上のプロセスが相互に通信を行っていることを意味する。分散システムにおいて、異なるコンピュータ上で実行するプロセスは共通の目的を達成するためにしばしばお互いに通信する必要がある。例えば、分散システムの各コンピュータはそれぞれのローカルの資源の利用状況を監視するために**資源管理 (resource manager)** のプロセスを持ち、これらの資源管理のプロセスはシステム全体の負荷調整を動的に行うために定期的に通信を行う。したがって、分散オペレーティングシステムはこのような通信機能のためにプロセス間通信 (IPC) 機構を提供する必要がある。

プロセス間通信 (interprocess communication) は、基本的に複数のプロセスにおける情報の共有を要求する。情報の共有のための基本的な2つの方法として次のものがある。

1. そのものを共有、共有データ方式
2. コピーを共有、メッセージパッシング方式

共有データ方式において、共有する情報はIPCに含まれる全てのプロセスがアクセスできる共通メモリに置かれる。この共有データ方式は図3.1(a)に示す概念的な通信形態をとる。一方、メッセージパッシング方式では共有する情

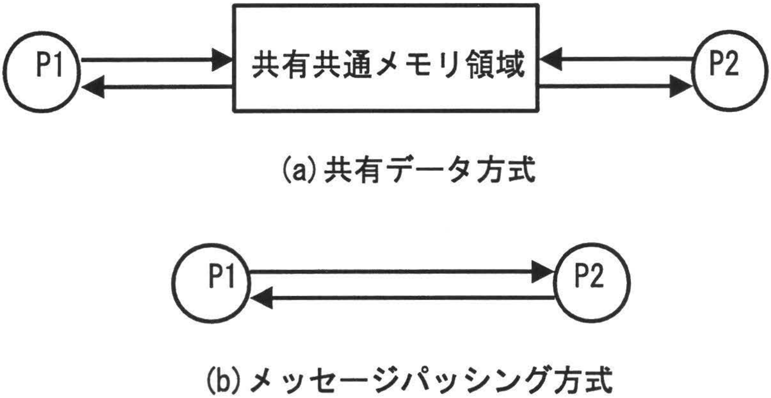


図 3.1　基本的な 2 つのプロセス間通信方式

報は物理的に送信プロセスプロセスのアドレス空間から全ての受信プロセスのアドレス空間にコピーされ、コピーされたデータはメッセージ (メッセージ (**message**) は情報のブロックである) の形式で転送される。メッセージパッシング方式は図 3.1(b) に示す概念的な通信形態をとる。すなわち、通信するプロセスは相互に直接作用する。

ネットワークのコンピュータはメモリを共有しないため、分散システムのプロセスは通常共有データではなくメッセージの交信によって通信を行う。したがって、メッセージパッシングは分散システムにおける基本的な IPC 機構である。

メッセージパッシングシステム (**message-passing system**) は、メッセージに基づく IPC プロトコルの集合を提供し、複雑なネットワークプロトコルや複数の不均質なプラットフォームの詳細をプログラマに見えないようにする分散オペレーティングシステムのサブシステムである。このことにより、プロセスはメッセージを交信することで通信ができ、プログラムを簡単な通信プリミティブ、例えば send や receive を使って書くことができる。これは、他の高位レベルの IPC システム、例えば**遠隔手続き呼び出し (remote procedure call)**(RPC、第 4 章参照) や分散共用メモリ (DSM、第 5 章参照) を構築するための適切な基盤となる。

3.2 良いメッセージパッシングシステムに望まれる機能

3.2.1 単純性

メッセージパッシングシステムは、簡単で使いやすいことが望まれる。メッセージパッシングシステムが提供するプリミティブを使うことによって新しいアプリケーションを構築したり、既存のアプリケーションとの通信が簡単にできなければならない。また、プログラマにとってアプリケーションのレベルでシステムやネットワークの条件を意識することなく可能な限り簡単な方法で分散アプリケーションの各種のモジュールを設計でき、それらのモジュール間でのメッセージの送信や受信ができることが必要である。メッセージパッシングシステムの明瞭で簡単な機能定義は分散アプリケーションの構築を容易にし、かつ間違いを起こさない。

3.2.2 統一的なセマンティクス

分散システムにおいて、メッセージパッシングシステムは次の2つのプロセス間通信のタイプに使用される。

1. **ローカル通信 (local communication)**:同じノード内のプロセス間での通信
2. **遠隔通信 (remote communication)**:異なるノードのプロセス間での通信

メッセージパッシングシステムの設計における重要な課題は、ローカル通信の意味と遠隔通信の意味が可能な限り一致していることである。このことは、メッセージパッシングシステムが容易に使用できるための重要な要求仕様である。

3.2.3 有効性

有効性は、一般にユーザがメッセージパッシングシステムを受け入れる上で大切な問題である。メッセージパッシングシステムが有効でなければ、プロセス間通信は無駄になり、アプリケーションの設計者はアプリケーションにおい

てこの使用を避けようとする。その結果、開発されたアプリケーションプログラムは捨てられることになる。メッセージパッシングのIPCプロトコルは、プロセス間通信において可能な限りメッセージの交信の数を少なくすることによって実際に有効となるようにすることができる。一般に、次のような方法が有効性を高めるための最適化方法としてある。

■同じプロセス対における各メッセージ毎のコネクションの設定と切断の回避

■コネクションの管理の最小化

■送信プロセスと受信プロセス間で交信されるメッセージへの前のメッセージの応答の重畳

3.2.4 信頼性

分散システムは、ノードのクラッシュや通信回線の障害などの各種の破滅的な事柄を受けやすい。このような障害は、2つのプロセスが通信している最中に発生し、メッセージの紛失などを引き起こす。**高信頼 (reliability)**IPCプロトコルは、障害の問題にも対応でき、メッセージの配信を保証する。メッセージの紛失に対する処理は、応答やタイムアウトによる再送によって通常行われる。

信頼性に関わる別の課題として、メッセージの重複がある。メッセージの重複は、障害やタイムアウトでの送信によって起きる。このための処理は、メッセージに順序番号を付与することで通常行われる。

良いメッセージパッシングシステムは、このような信頼性の高いIPCプロトコルを持たなければならない。

3.2.5 正確性

メッセージパッシングシステムは、送信プロセスから受信のグループにメッセージを送信し、又は複数の送信プロセスからメッセージを受信するようなグループ通信を可能とするIPCプロトコルを持つ。**正確性 (correctness)** は、このグループ通信のIPCプロトコルに関連する機能である。いつも要求されるものではないが、あるアプリケーションに対して有効である。この正確性に関する課題については、下記のものがある [Navratnam et al 1988]。

■**原子性 (atomicity)**
■**順序配信 (ordered delivery)**
■**生存性 (survivability)**

原子性は、受信グループに送信されるメッセージが受信者全てに送られるか若しくは誰にも送られないかを保証するものである。順序配信は、全ての受信者に届くメッセージが正しい順序でアプリケーションに届けられることを保証することである。生存性は、メッセージがプロセス、マシン又は通信リンクの障害に対しても正しく配信されることを保証することである。この生存性は達成するには困難な特性である。

3.2.6　柔軟性

全てのアプリケーションが、IPC に対して同じレベルの信頼性や正確性を要求するものではない。例えば、適応中継方式において、ネットワークの各場所でキューの遅延に関する情報を伝えることが必要になるかもしれない。しかしながら、通信の障害のためにブロードキャストメッセージが遅れた場合、この情報は届かなかったものと同じ意味になる。同様に、多くのアプリケーションはメッセージの原子性や順序配信を必要としない。例えば、クライアントはサーバのグループに要求をブロードキャストし、最初に応えるサーバに仕事を依頼する。このような場合、確かにメッセージの配信の原子性は要求されない。よって、メッセージパッシングシステムの IPC プロトコルは、アプリケーションの多様な要求に対応できるように十分柔軟である必要がある。すなわち、IPC プリミティブはアプリケーションの要求する信頼性や正確性のタイプやレベルを選択し、かつ指定できるような**柔軟性 (flexibility)** を持つことである。さらに、IPC プリミティブはプロセス間の各種のフロー制御を同期及び非同期の Send/Receive に関連して可能にできる柔軟性を持たなければならない。

3.2.7　セキュリティ

良いメッセージパッシングシステムは、安全なエンドツーエンドの通信を提供する能力も必要である。すなわち、ネットワーク上のメッセージは宛先と送信者以外の誰からもアクセスされてはいけない。**セキュリティ (security)** のある通信には次のことが必要となる。

■送信者によるメッセージ受信者の認証
■受信者によるメッセージ送信者の認証
■ネットワーク送信時のメッセージの暗号化

これらの課題については第 11 章で詳細に説明する。

3.2.8　移植性

メッセージパッシングシステムにおける**移植性 (portability)** には 2 つの特徴がある。

1. メッセージパッシングシステム自身が移植性を持つ。すなわち、既存のメッセージパッシングシステムの基本的な設計を再利用することで他のシステムに新しい IPC 機能を容易に構築できる。

2. メッセージパッシングシステムの IPC プロトコルのプリミティブを利用して作成されたアプリケーションは移植性を持つべきである。これは、メッセージパッシングシステムの設計おいて不均質性を考慮に入れることを要求している。すなわち、異なるアーキテクチャのコンピュータ上で動作するプロセス間での通信においてデータ表現形式が陽に定義されていることが必要となる。メッセージパッシングシステムの IPC プロトコルに対する高位レベルのプリミティブ設計は、ネットワークの不均質性を隠蔽するように行わなければならない。

3.3　メッセージパッシングの IPC の課題

メッセージは、受信プロセスで理解できるように送信プロセスが形式化した情報のブロックである。これは、固定長のヘッダと可変長の形式化されたデータオブジェクトの集合からなる。図 3.2 に示すように、ヘッダは次の要素から構成される。

■**アドレス (address)**:これはネットワークの送信プロセスと受信プロセスを識別するための文字を含む。これは 2 つの部分、送信プロセスのアドレスと受信プロセスのアドレスからなる。

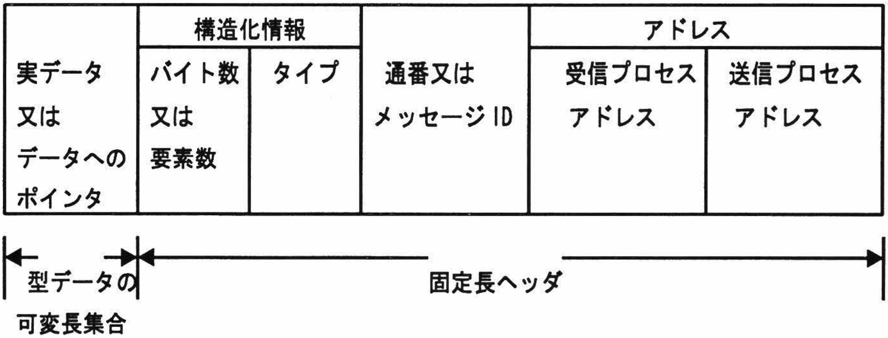


図 **3.2**　典型的なメッセージ構造

■順序番号 (sequence number):これはメッセージの識別子 (ID) であり、システム障害で紛失や重複したメッセージを識別するために有効となる。

■構造化情報 (structural information):この要素は 2 つの部分からなる。型は受信プロセスに渡されるデータがメッセージに含まれているか、又はメッセージ外にある場合のデータへのポインタを含んでいるかを規定している」。2 番目の部分は、可変長のメッセージデータの長さを示している。

メッセージに基づく IPC プロトコルにおいて、送信プロセスはメッセージの実際の内容を決定し、受信プロセスはその内容の解釈の方法を知っている。特別のプリミティブがメッセージの送信と受信に対して明示的に使用される。したがって、この方法において、ユーザは通信で使用されるメッセージ形式やメッセージの送信及び受信の機構を十分に知っている。

メッセージパッシングシステムの IPC プロトコルの設計において、次のような重要な課題を検討する必要がある。

■送信者は誰か？
■受信者は誰か？
■受信者一人か複数か？
■メッセージは受信されたとき保証されているか？

■送信者は応答を待つ必要があるか？

■通信中にノードのクラッシュや通信リンクの障害などが発生した場合どうなるか？

■受信プロセスが用意できてない時、メッセージは廃棄されるか、又はバッファに保持されるか？ バッファがない時はどうなるか？

■受信プロセスに渡すメッセージがある場合、どのような順番で渡せば良いか？

これらの課題は、IPCプロトコルによって提供される通信プリミティブの集合の動作の定義によって行われる。メッセージに基づくIPCプロトコルのこれらの課題について次に説明する。

3.4 同期

通信における中心的な課題は、通信プリミティブによる通信プロセスに負わされている**同期 (synchronization)** の問題である。同期の意味として、大きくブロッキング **(blocking)** と非ブロッキング **(nonblocking)** の2つのタイプに分類される。呼び出し側の実行をブロックしない(呼び出し側に制御もすぐに戻される)プリミティブは、非ブロッキングの意味を持つ。それ以外に、ブロッキングタイプのプリミティブがある。通信プロセスに関係する同期は、基本的にsendとreceiveプリミティブの使用上の意味としてこれらの2つのタイプのいずれかである。

ブロッキングのsendプリミティブの場合、send文を実行後、送信プロセスは受信プロセスがメッセージを受信したことの応答を受けるまでブロックされる。一方、非ブロッキングのsendプリミティブでは、送信プロセスはメッセージがバッファにコピーされると直ちに実行できる状態になる。

ブロッキングのreceiveプリミティブの場合、受信プロセスはreceive文を実行して、メッセージを受信するまでブロックされる。一方、非ブロッキングのreceiveでは、受信プロセスはreceive文を実行するとメッセージバッファがあるカーネルの指示した後制御を即座に返され、次の実行に移る。

非ブロッキングのreceiveプリミティブの重要な課題は、受信プロセスがメッセージバッファに到着したメッセージをどのように知るかである。このための方法として次の2つがある。

1. **ポーリング (polling)**:この方法では、バッファの状態を調べるための test プリミティブが受信プロセスに提供される。受信プロセスは定期的にこのプリミティブを使ってメッセージが到着しているかをカーネルに問い合わせる。

2. **割り込み (interrupt)**:この方法では、メッセージが到着し、受信プロセスが受け取れる状態になると、ソフトウェア割り込みによって受信プロセスに通知される。この方法は、受信プロセスが何度も test プリミティブを出すことなく実行を継続できる。この方法は効果的であり、最大の並列処理を実現するが、欠点としてユーザレベルの割り込みによるプログラミングの複雑さを持つ [Tanenbaum 1995]。

非ブロッキングの receive プリミティブの変形として、従来の receive プリミティブがあり、同じように呼び出したプロセスに制御が即座に返され、かつメッセージが到着しているか否かを通知する。

ブロッキングの send プリミティブでは、送信プロセスは受信プロセスが障害を起こしたり、通信障害のためにネットワークでメッセージが受信されなければいつまでもブロッキングされた状態が続く。この状態を防ぐために、ブロッキングの send プリミティブでは send 動作が指定した時間を経過しても完了しない場合には誤り状態として通知するタイムアウトを使用する。タイムアウトの値は、既定値か若しくは柔軟性を持たせるために send プリミティブのパラメータの 1 つとしてユーザが指定したものである。

タイムアウト値は、同様にブロッキングの receive プリミティブについても受信プロセスがブロッキング状態から抜けるために使用される。

2 つのプロセス間通信において send と receive プリミティブがブロッキングの意味で使用された場合、この通信を同期と呼ぶ。もう一方を非同期と呼ぶ。すなわち、**同期通信 (synchronous communication)** では送信プロセスと受信プロセスはメッセージを交信することで同期をとらなければならない。この様子を図 3.3 に示す。基本的な動作として、送信プロセスはメッセージを受信プロセスにメッセージを送り、応答を待つ。receive プリミティブ文を実行すると受信プロセスはメッセージを受信するまでブロックされる。メッセージを受信すると、受信プロセスは送信プロセスに応答メッセージを返す。送信プロセスは、この応答メッセージを受信した時に実行を再開できる。

非同期通信 (asynchronous communication) に比べて、同期通信は実現が簡単で容易である。また、同期通信は送信プロセスがメッセージを受信して

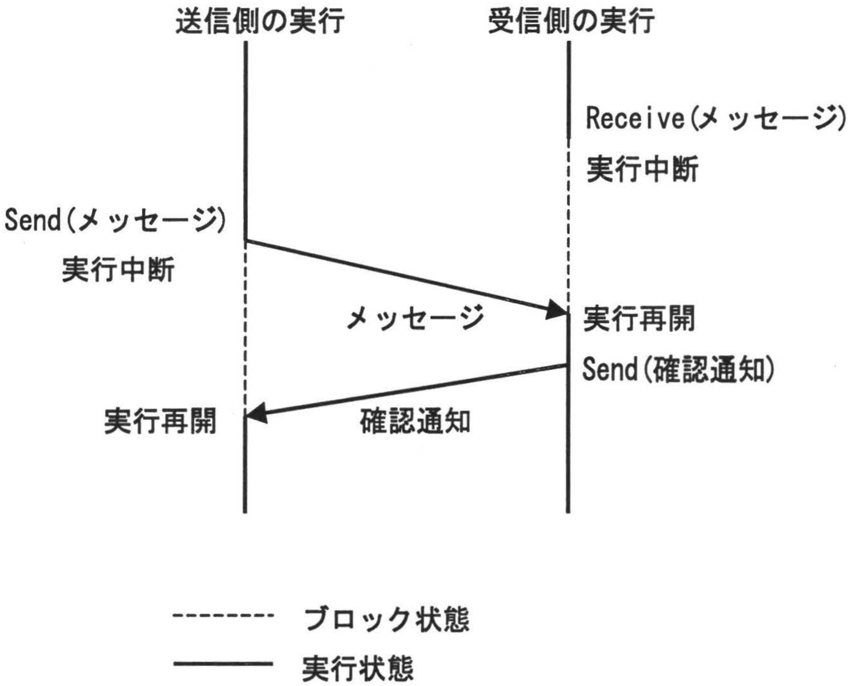


図 **3.3** ブロッキングタイプの send と receive プリミティブによる同期モードの通信

から次の実行に進むため、信頼性が高い。このため、メッセージが正しく配信されない場合、送信プロセスは状態の一貫性を確認して実行するための誤り再開処理が必要とされない [Shatz 1984]。しかしながら、同期通信の主な欠点は、並行性が制限され、通信のデッドロックの問題を含むことにある (通信のデッドロックについては第 6 章で説明する)。この同期通信は、必要でない場合にも受信プロセスからの応答を送信プロセスが必ず持たなければため、非同期通信に比べて柔軟性に劣る。単一プロセスで多重スレッドを実現するシステムでは (第 8 章参照)、ブロッキングのプリミティブが制限される並行性の欠点が現れることなく使用できる。どのようにこれが行われるかは第 8 章で説明する。

柔軟性のあるメッセージパッシングシステムは、send と receive に対してブロッキングと非ブロッキングの両者のプリミティブを提供し、ユーザはアプリケーションの要求に合うものを選ぶことができる。

3.5　バッファリング

メッセージは、あるプロセスから他のプロセスに送信プロセスのアドレス空間から受信プロセスのアドレス空間に(送信及び受信するコンピュータのカーネルのアドレス空間を介して)メッセージの本体をコピーすることで転送できる。しかし、受信プロセスは転送されたメッセージを受信する用意ができてないことあり、後に受信するためにオペレーティングシステムにメッセージを保持することを要求する。このような場合、オペレーティングシステムは受信プロセスがメッセージを受信して実行する前にメッセージを保持するためのバッファを必要とする。

プロセス間通信において、メッセージバッファリング方式は同期方式に強く関係する。通信における同期方式と非同期方式はそれぞれバッファリングの2つの方式、**ヌルバッファ (null buffer)** 又は**無バッファリング (no buffering)**、と**無制限バッファ (buffer with unbounded capacity)** に対応する。バッファリングについて他の2つの方式として、**単一メッセージ (single-message)** と**有限メッセージ (finite-bound)** 又は**多重メッセージ (multiple-message)** がある。これらの4つのバッファリング方式について次に説明する。

3.5.1　ヌルバッファ(又は無バッファリング)

この場合、メッセージの一時的に保持するバッファはない。したがって、次のような実現方法が採られる。

1. メッセージは受信プロセスの receive が実行されるまで、send の実行は待たされ、メッセージは送信プロセスのアドレス空間に置かれる。このために、送信プロセスのブロックが解け、send 文を実行開始するまで送信プロセスは中断される。受信プロセスが receive を実行すると、送信プロセスのカーネルに応答が送られ、カーネルは送信プロセスからメッセージを送信できることを通知する。応答メッセージを受けると、送信プロセスはブロックが解け、send の実行を再開できる。この時点で、メッセージは受信プロセスが受信状態になっているため送信プロセスのアドレス空間から受信プロセスのアドレス空間に正しく送られる。

2. メッセージは単に捨てられ、一定時間後にメッセージを再送するためにタ

イムアウト機構が機能する。すなわち、send を実行した後、送信プロセスは受信プロセスからの応答を待つ。応答がタイムアウトの時間内に受けられなければ、メッセージが捨てられたものとして受信が行われるために再試行する。成功するまでに送信プロセスは何度か行うこともある。特定回数送信後、送信プロセスは送信をあきらめる。

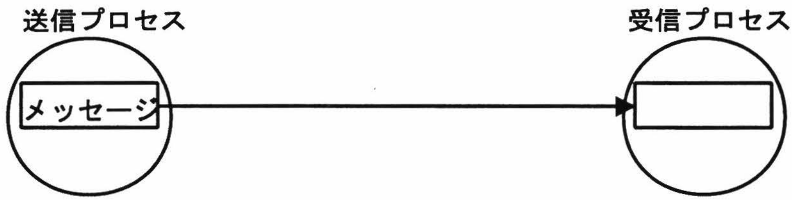
図 3.4(a) に示すように、無バッファリングの場合、メッセージ転送の論理パスは 1 つのコピー動作を行うために送信プロセスのアドレス空間から受信プロセスのアドレス空間に直接設定される。

3.5.2 単一メッセージバッファ

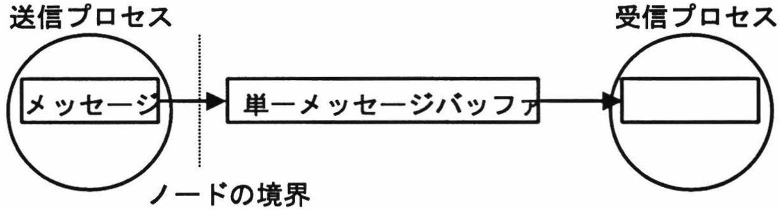
ヌルバッファ方式は、受信プロセスが用意できていない場合、メッセージを複数回転送しなければならないし、メッセージの受信プロセスはメッセージがネットワークを転送される時間を待たなければならないため、分散システムにおける 2 つのプロセス間での同期通信には適さない。分散システムでは、ネットワークでのメッセージの転送時間がある場合には重要になる。したがって、ヌルバッファ方式の代わりにネットワーク若しくは分散システムの同期通信では単一メッセージバッファ方式を採用する。この方式では、1 つのメッセージを保持できる容量のバッファが受信プロセスのノードに用意される。これは、同期通信に基づくシステムではアプリケーションモジュールはある時点で多くても 1 つの待たされるメッセージを持つと考えているためである。この単一メッセージバッファ方式の基本的な考え方は、受信プロセスに 1 つのメッセージを受けるための用意することである。したがって、この方式では受信プロセスがメッセージを受ける用意ができていない場合、要求メッセージ受信プロセスのノードのバッファに蓄積される。メッセージバッファは、カーネルのアドレス空間か受信プロセスのアドレス空間に配置される。図 3.4(b) に示すように、この場合、2 つのコピー動作を含む論理パスが存在する。

3.5.3 無制限容量のバッファ

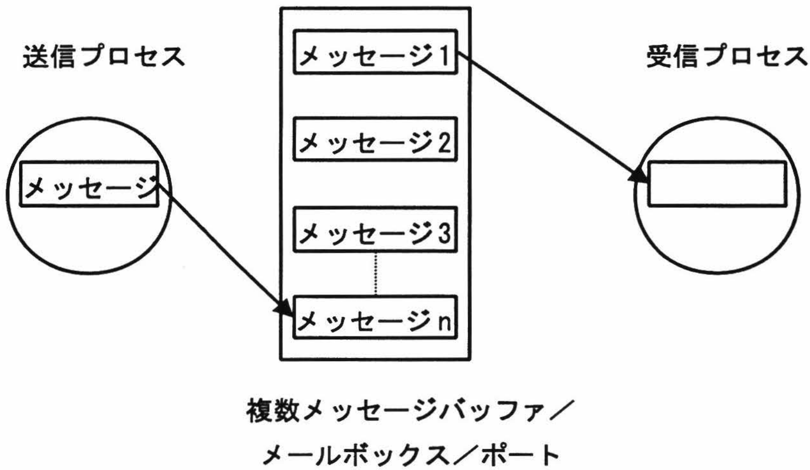
非同期モードの通信においては、受信プロセスが用意できているかについて送信側は待たないため、受信プロセスが受信できるかを確認するための幾つかのメッセージが発生する。したがって、受信側に送られる全てのメッセージの



(a) 無バッファ方式による同期送信のメッセージ転送
(1 回のコピーが必要)



(b) 単一メッセージバッファ方式による同期送信のメッセージ転送
(2 回のコピーが必要)



(c) 複数メッセージバッファ方式による非同期送信のメッセージ転送
(2 回のコピーが必要)

図 **3.4**　プロセス間通信機構における 3 種類のバッファリング方式

送達を保証するために、非同期通信では受信できないメッセージを一時的に蓄えるための無制限容量のメッセージバッファが必要となる。

3.5.4 有限（又は多重メッセージ）バッファ

バッファを無制限に持つことは現実的には不可能である。したがって、実際の非同期モードの通信のシステムでは有限のバッファを使用し、これは多重メッセージバッファとも呼ばれる。バッファが有限であるため、この方式ではバッファオーバフローについての問題を考えなければならない。バッファオーバフローの問題は、次の2つの方法で対応される。

1. **不成功通信 (unsuccessful communication)**:この方式では、メッセージ転送はバッファがなくなると単に失敗する。send は送信プロセスにバッファがないためにメッセージが受信できないことを示す誤りメッセージを返す。しかしながら、この方式は信頼性の低いメッセージパッシングである。

2. **フロー制御通信 (flow-controlled communication)**:この方法は、フロー制御を使用し、受信側がメッセージを受信して次のメッセージを受信するためのバッファを準備できるまで送信側をブロックする。この方法は、送信プロセスと受信プロセスの間の同期をとり、期待しないデッドロックを生じさせることもある。されに、同期を行うため非同期の送信は、全ての send コマンドにおいて実際に非同期モードでは動作しない。

有限なバッファ方式で割り当てられるバッファの大きさは、実装に依存する。多くの場合、ユーザに対して create_buffer システム呼び出しが提供される。このシステム呼び出しが受信プロセスによって実行されると、指定された大きさのバッファが生成される (時には**メールボックス (mail box)** 又は**ポート (port)** と呼ばれる)。受信プロセスのメールボックスは、カーネルのアドレス空間か受信プロセスのアドレス空間に配置される。カーネルのアドレス空間に置かれる場合、メールボックスは要求したプロセスに割り当てられるシステムの資源である。個々のプロセスがそのメールボックスに保持できるメッセージの数には制限がある。一方、メールボックスが受信プロセスのアドレス空間に置かれる場合、オペレーティングシステムはプロセスにメモリの割り当てや障害などへの保護を任せる。

図 3.4(c) に示すように、有限のバッファ方式における非同期の send の場合、メッセージはまず送信プロセスのメモリから受信プロセスのメールボックスにコピーされ、受信プロセスがメッセージの受信のコールを行った時にメールボックスから受信プロセスのメモリにコピーされる。したがって、この場合もメッセージ転送の論理パスは 2 つのコピー動作を含む。

多重メッセージバッファリング機能に基づくメッセージ通信は、無バッファリングや単一メッセージバッファリングに比べて高い並行性と柔軟性を提供するが、設計や使用方法が複雑になる。これは、バッファ管理における生成、削除、保護及びその他の問題に必要となる機構に新たな処理やオーバヘッドを含むためである。

3.6　マルチデータグラムメッセージ

ほとんど全てのネットワークは、一度に転送できるデータ長の上限を持つ。この大きさは、ネットワークの最大転送単位 **(maximum transfer unit)(MTU)** と呼ばれる。MTU を越えるメッセージは、複数の MTU に分割され、別々に送られなければならない。各セグメントは、メッセージデータに加えてある制御情報を持つパケットとして送られる。各パケットは**データグラム (datagram)** と呼ばれる。ネットワークの MTU 以下のメッセージは、1 つのパケットとして送られ、**単一データグラムメッセージ (single-datagram message)** と呼ばれる。一方、ネットワークの MTU を越えるメッセージは分割され、複数のパケットとして送られる。このようなメッセージは、**マルチデータグラムメッセージ (multidatagram message)** と呼ばれる。明らかに、マルチデータグラムメッセージの各パケットは他のパケットと順序的な関係を持っている。送信側でマルチデータグラムメッセージを複数のパケットに分割し、受信プロセスでパケットを組み立てることは通常メッセージパッシングシステムの責任である。

3.7　メッセージデータの符号化と復号化

メッセージデータは、受信プロセスにとって理解できるものでなければならない。すなわち、理想的にはメッセージが送信プロセスのアドレス空間から受信プロセスのアドレス空間に転送された後にもプログラムオブジェクト構造が保持されることである。このことは、異なるアーキテクチャのコンピュータ上

に存在する送信プロセスと受信プロセスからなる不均質なシステムでは明らかに無理である。しかしながら、次の2つの理由から均質なシステムにおいてさえも実現することは大変困難である。

1. あるプロセスのアドレス空間から別の空間に転送されると、ポインタの絶対値はその意味を持たなくなる。したがって、絶対値のポインタを使用するこのようなプログラムオブジェクトは、そのままの形式では転送できなく、転送するために別の表現形式を採らなければならない。例えば、木構造のオブジェクトを転送するためには、他のプロセスに転送する前に木構造の各要素をリーフレコードとしてコピーし、バッファにある決まった順番に並べかえる必要がある。リーフレコード自身は、受信プロセスのアドレス空間では意味を持たないが、それから容易に木構造が再生できる。このような再生の機能のために、オブジェクト型情報は送信側と受信側の間で渡され、木構造だけでなくリーフレコードの並び順の情報まで示されなければならない。木構造の平面化や再構成の処理は、例えば関係付けられたリストのような他の構造化プログラムオブジェクトに対しても拡張できる。

2. 各種のプログラムオブジェクトは、大きな格納領域を必要とする。意味を明確にするために、メッセージは一般に各種のプログラムオブジェクト型、例えば長整数型、短変数型、可変長文字列型などを含む。この場合、受信プロセスにメッセージが理解できるためにプログラムオブジェクトが置かれるバッファの位置や各プログラムオブジェクトの領域の大きさを識別できる方法が必要である。

プログラムのオブジェクトをそのままの形式で転送すると述べたような問題があるため、それらはまず転送に適したストリーム形式に変換され、メッセージバッファに置かれる。この変換処理は、送信プロセスで行われ、メッセージデータの**符号化 (encoding)** と呼ばれる。受信プロセスで受信された符号化されたメッセージは、ストリーム形式から元のプロブラムオブジェクトに戻されなければならない。受信プロセスでのメッセージデータからプログラムオブジェクトに再構成する処理は、メッセージデータの**復号化 (decoding)** と呼ばれる。

メッセージデータ符号化と復号化で使用される表現形式として次の2つの方法が用いられる。

1. **タグ付き表現 (tagged representation)** では、値を持った各プログラムオブジェクトの型はメッセージ内に符号化される。この方法では、符号化データ形式の自己記述性のために受信プロセスにとってメッセージ内の各プログラムオブジェクトの型を調べることは容易である。

2. **タグ無表現 (untagged representation)** では、メッセージデータはプログラムオブジェクトのみを含む。各プログラムオブジェクトの型を指定する情報は、メッセージデータには何もない。この方法では、符号化されるデータ形式が自己記述性でないため、受信プロセスはあらかじめ受信したデータの復号化方法を知っていなければならない。

タグ無表現は、Sun の XDR(外部データ表現)[Sun 1990] と Courier[Xerox 1981] で使用されており、タグ付き表現は ASN.1(抽象構文記法) 標準 [CCITT 1985] と Mach 分散オペレーティングシステム [Fitzgerald and Rashid 1986] で使用されている。

一般に、タグ付き表現は、メッセージデータの符号化と復号化で要求される転送の品質及び処理時間に対して、タグ無表現よりも高価である。表現について何もなければ、送信プロセスと受信プロセスの両者はメッセージの符号化データの形式に十分注意をはらわなければならない。送信プロセスは符号化データ形式のために符号化ルーチンを処理し、受信プロセスは対応する復号化ルーチンを処理する。符号化と復号化の動作は、ローカルの表現を許容しながらも符号化されたデータを確かに復号化するという点では完全に対称である。時には、受信プロセスは全体が符号化されたデータ、例えば最大長を越えるデータを受信することもある。このような場合、受信プロセスは、受信データを正しく復号化できないためデータが理解できないことを示す誤りメッセージを送信プロセスに返す。

3.8　プロセスのアドレス付け

メッセージに基づく通信における別の重要な課題として、相互に作用する対のアドレス付け (又は名前付け) があり、送信プロセスはメッセージを誰に送るか、逆に受信側は誰からメッセージを受けるかを指定できなければいけない。より柔軟性を高めるために、メッセージパッシングシステムは通常 2 つのプロ

(a)**send**(process_id, message)

process_id で指定したプロセスにメッセージを送信

(b)**receive**(process_id, message)

process_id で指定したプロセスからメッセージを受信

(c)**send_any**(service_id, message)

service_id で指定したサービスを提供するプロセスにメッセージを送信

(d)**receive_any**(process_id, message)

任意のプロセスからメッセージを受信し、メッセージを受信したプロセスのプロセス識別子(process_id)を返す。

図 3.5 プロセスの明示的及び暗示的アドレス付けのプリミティブ

セスのアドレス付け方法に対応する。

1. 明示的アドレス付け **(explicit addressing)**:通信を要求されるプロセスは通信プリミティブのパラメータとして明示的に指定される。図 3.5 のプリミティブ (a) と (b) は明示的なプロセスのアドレス付けを要求する。

2. 暗示的アドレス付け **(implicit addressing)**:通信を望むプロセスは通信相手のプロセスを明示的に指定しない。図 3.5 のプリミティブ (c) と (d) は暗示的なプロセスアドレス付けを表している。プリミティブ (c) では、送信プロセスはプロセスの代わりにサービスを指定する。このプリミティブは、クライアントガ要求するサービスを実際に提供するサーバをクライアントが知らないようなクライアントサーバ通信に有効である。このようなプロセスのアドレス付けは、通信プリミティブで使用されるアドレスがプロセスよりもむしろサービスを識別するために**機能的アドレス付け (functional addressing)** とも呼ばれる。

一方、プリミティブ (d) では、受信プロセスはどの送信プロセスからもメッセージを受信する。このプリミティブも、サービスの利用を全てのクライアン

トに許可している場合にサーバは全てのクライアントからのサービス要求を受けることができるクライアントサーバ通信に有効である。

通信プリミティブで使用される基本的な 2 つのプロセスのアドレス付け方法において、共通的に使用される方法がある。

プロセスを識別するための単純な方法は、**machine_id** と **local_id** の組み合わせ、例えば **machine_id@local_id** である。**local_id** 部分はプロセス識別子、又は受信プロセスのポート識別子、又は装置内のプロセスをユニークに識別できる他の何かである。他のプロセスにメッセージを送ろうとするプロセスは、**machine_id@local_id** の形式で受信プロセスのアドレスを指定する。アドレスの machine_id 部分は、受信プロセスの装置にメッセージを送るために送信プロセスの装置のカーネルによって使用され、アドレスの **local_id** 部分は受信プロセスの装置のカーネルが指定されたプロセスにメッセージを渡すために使用する。このプロセスのアドレス付けの方法は、machine_id として 32 ビットのインターネットアドレスと **local_id** として 16 ビットの番号によってバークレイ版 UNIX で使用される。

この方法は、**local_id** が 1 つの装置内でユニークであればよく、他の装置との調整を必要としないため、プロセスをシステム全体でユニークにするための全体の管理が要求されない。しかしながら、この方式の欠点としてプロセスを別の装置に移す場合にはそのものではできない。例えば、システムのバランスを取るとために負荷の高い装置のプロセスを別の負荷の低い装置に移したい時に問題となる。

上記のような制限に対応するために、プロセスを次の 3 つの部分、**machine_id**、**local_id** および **machine_id** の組み合わせで識別する。

1. 最初の部分はプロセスが生成されたノードを表す。
2. 2 番目の部分はプロセスが生成されたノードによって与えられたローカルの識別子。
3. 3 番目はプロセスが最後いた場所 (ノード) を示す。

プロセスが生きている間、識別子の最初の 2 つの値は変更されないが、最後の部分は変わる。このプロセスのアドレス付け方法は、**リンクに基づくプロセスアドレス付け (link-based process addressing)** と呼ばれる。この方法が正しく機能するためには、プロセスが新しいノードに移動した時、以前のノードにリンク情報 (新しいノードの machine_id に等しい 3 番目の部分の値を持つ

プロセス識別子) が残され、新しいノードではプロセスに新しい **local_id** は与えられ、他のノードで生成された全てのプロセスに対して新しいノードのカーネルによって管理されるマッピングテーブルに登録される。プロセスが生成された時は、プロセス識別子の 3 番目の値は最初の部分の値と等しい。

他のプロセスにメッセージを送信したいプロセスは、受信プロセスのアドレスを **machine _id@local_id@machine_id** の形式で指定する。送信プロセスの装置のカーネルは、受信プロセスのアドレスの 3 番目に指定されている **machine_id** の装置にメッセージを送信する。3 番目の値が 1 番目の値と同じであれば、メッセージはプロセスが生成されたノードに送られる。受信プロセスが移動していなければ、プロセス識別子の **local_id** 情報を使ってメッセージは送られる。一方、受信プロセスが移動していれば、以前のノードに残されているリンク情報によって受信プロセスが移動した先のノードにメッセージが送られる。この方法によって、メッセージは受信プロセスの現在のノードに到達するまで何度かノードを中継転送される。メッセージが受信プロセスの現在いるノードに届くと、そのノードのカーネルはメッセージの送信プロセスと受信プロセスのプロセス識別子を調べる。受信プロセスのプロセス識別子の最初の 2 つは、マッピングテーブルの **local_id** と確認するためにユニークな識別子として使用され、指定されたプロセスにメッセージを渡すことができる。一方、送信プロセスのプロセス識別子は受信プロセスの現在の場所を送信プロセスに通知するために使用される。送信プロセスは、この情報から受信プロセスの 3 番目の値を更新し、その装置のローカルキャッシュに保持し、次にその受信プロセスにメッセージを送る場合にはプロセスが生成されたノードを経由しないで直接受信プロセスがいるノードにメッセージを送る。このプロセスアドレス付け方法に類似したものとして、DEMOS /MP[Miller et al. 1987] と Charlotte[Artsy et al. 1987] がある。この方法はプロセスの移動に対して有効であるが、次の 2 つの欠点がある。

1. プロセスの移動が多いとプロセスの位置を確定するためのオーバヘットが大きい。
2. プロセスが以前いたノードが障害となるとプロセスの位置の確定ができなくなる。

既に述べたプロセスアドレス付け方法は、装置の識別子を指定する必要があるため透過性を持たない。ユーザは、プロセスの位置を気にしなければならな

い(又は少なくともプロセスが生成された場所)。しかしながら、第1章で位置透過性は分散オペレーティングシステムの主要な目的の1つであることを説明した。したがって、位置透過性を持つプロセスアドレス付け機構がメッセージパッシングシステムには強く要求される。この目的を達成するための単純な方法は、プロセスをシステム全体でユニークにする識別子に装置の識別子を含まないことである。カウンタを管理する集中型のプロセス識別子割り当てをこのために使用することができる。識別子のための要求を受信すると、カウンタの現在の値を単に返し、カウンタに1加算する。しかし、この方法は信頼性と規模拡張性に問題を持っている。

プロセスアドレス付けの**位置透過性 (location transparency)** を達成する別の方法として、プロセスに対して2つのレベルの名前付けを行う方法である。この方法では、各プロセスは2つの識別子、装置に依存しない高位レベルな名前(ASCII 文字列)と装置に依存する低位レベルな名前(例えば **machine_id@local_id**)を持つ。**名前サーバ (name server)** は、プロセスの高位レベルの名前と低位レベルの名前の対応付けるマッピングテーブルを管理する。この方法では、他のプロセスにメッセージを送りたいプロセスは通信プリミティブで受信プロセスの高位レベルの名前を指定する。送信プロセスの装置のカーネルは、まず高位レベルの名前から受信プロセスの低位レベルの名前を得るために名前サーバ(全ての装置はアドレスを知っている)に接触する。低位レベルの名前から、カーネルは指定された装置にメッセージを送信し、受信プロセスのカーネルはメッセージを受信プロセスに渡す。送信カーネルは、次のために受信プロセスの高位レベルの名前と低位レベルの名前の対応情報をローカルのキャッシュに保持し、次の送信時には名前サーバとは交信をは行わない。

この名前サーバの方法は、通信を行うプロセスのコードの変更なしでプロセスの移動を可能にする。これは、プロセスの低位レベルの名前の変更が起きた場合、名前サーバのマッピングテーブルの更新で済むためである。しかし、プロセスの高位レベルの名前は保持することが必要である。

名前サーバ方式は、機能的なアドレス付けに対しても適している。この場合、高位レベルの名前はプロセスに代えてサービスを識別し、名前サーバはサービス識別子をサービスを提供する1つ若しくは複数のプロセスに対応させる。

名前サーバ方式も、名前サーバがシステムの集中的な構成要素であるために信頼性と規模拡張性に問題がある。この問題を解決するためには、名前サーバの多重化が必要である。しかしながら、これは多重化における一貫性を確保するために大きなオーバヘッドが発生する。

3.9 障害処理

分散システムは並列処理の機能を提供するが、ノードの故障や通信リンクの障害などの部分的な障害の影響を受け易い。図 3.6 に示すように、プロセス間通信においてこのような障害は次のような問題を引き起こす。

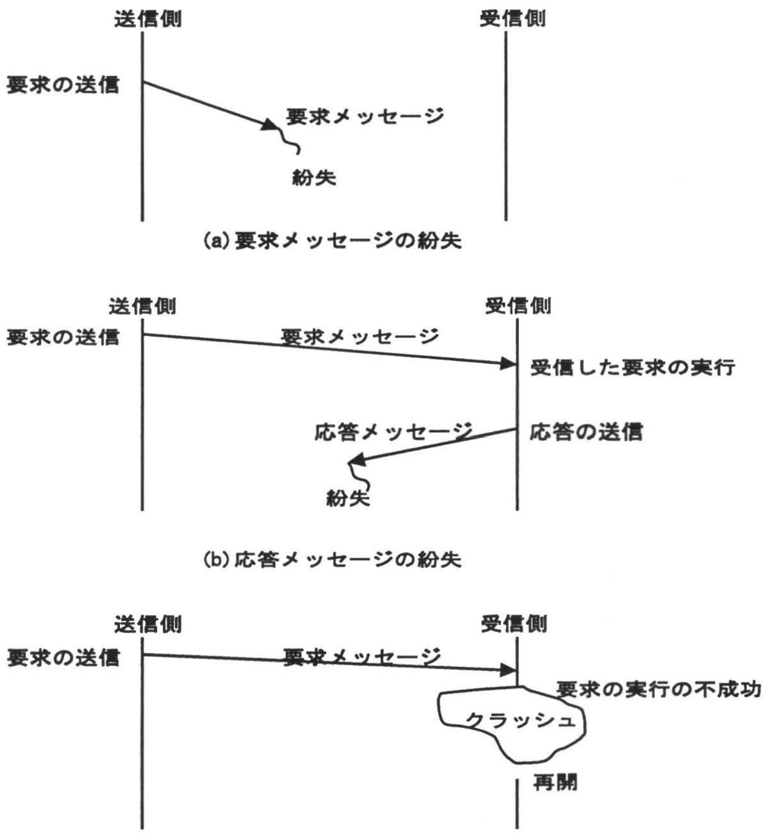


図 **3.6** システム障害による IPC の問題

1. 要求メッセージの紛失。これは送信プロセスと受信プロセスの間の通信リンクの障害若しくは要求メッセージが受信ノードに届いた時に障害になっていることが原因である。

2. 応答メッセージの紛失。これは送信プロセスと受信プロセスの間の通信リンクの障害若しくは応答メッセージが送信ノードに届いた時に障害になっていることが原因である。

3. 要求の実行の不成功。これは要求を処理している時に受信ノードがクラッシュすると起きる。

このような問題に対応するために、メッセージパッシングシステムの信頼性のある IPC プロトコルは、タイムアウトによるメッセージの再送や受信装置のカーネルから送信装置のカーネルへの応答メッセージの返送の考え方に基づいて設計される。すなわち、送信装置のカーネルは受信装置から確認通知がなければタイムアウトによってメッセージの再送を行う責任を持つ。送信装置のカーネルは、確認通知が受信された場合のみ送信プロセスを解放する。送信プロセスが要求を再送するために待つ時間は、送信プロセスと受信プロセスの間のラウンドトリップ時間に要求を実行するために必要となる平均的な時間を加えた時間より少し大きくなる。

このような考え方に基づいて、2つのプロセス間でのクライアントサーバ通信に対する4メッセージ信頼性 IPC プロトコルは次のように動作する (図 3.7 参照)。

1. クライアントはサーバに要求メッセージを送る。

2. 要求メッセージをサーバ装置で受信すると、その装置のカーネルは確認通知メッセージをクライアントの装置のカーネルに返す。確認通知がタイムアウト時間内に返されなければ、クライアントの装置のカーネルは要求メッセージを再送する。

3. サーバがクライアントの要求の処理を完了すると、クライアントに応答メッセージ (処理結果を含む) を返す。

4. 応答メッセージがクライアントの装置に届くと、それに対する確認通知メッセージをその装置のカーネルはサーバの装置のカーネルに返す。確認通知メッセージがタイムアウト時間内に受け取らなければ、サーバは応答メッセージを再送する。

クライアントサーバ通信では、処理された要求の結果は要求メッセージをサーバが受信したことの確認通知の意味も含む。この考え方による2つのプロセス間のクライアントサーバ通信に対する3メッセージ信頼性 IPC プロトコル

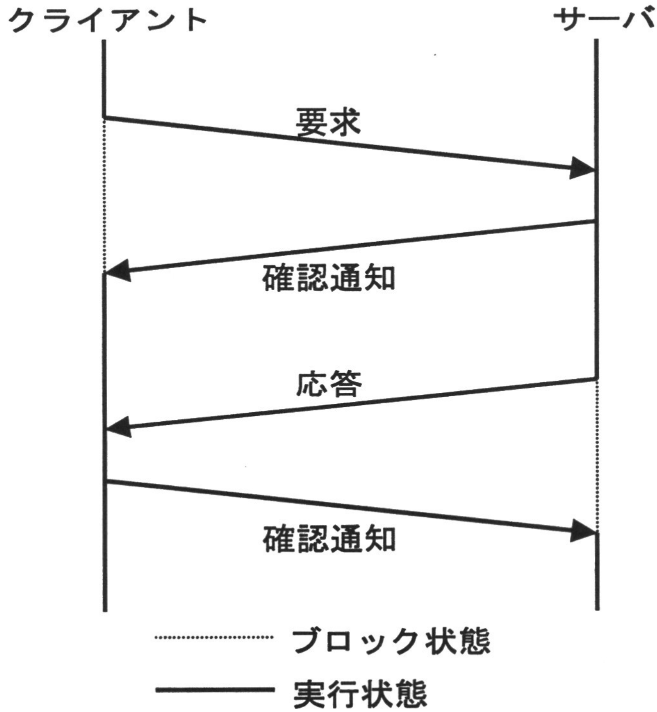


図 3.7 2つのプロセス間でのクライアントサーバ通信の 4 メッセージ高信頼 IPC プロトコル

は次のように動作する (図 3.8 参照)。

1. クライアントはサーバに要求メッセージを送る。

2. サーバがクライアントの要求の処理を完了すると、クライアントに応答メッセージ (処理結果を含む) を返す。クライアントは応答が返るまでブロックされる。応答がタイムアウト時間内に返されなければ、クライアントの装置のカーネルは要求メッセージを再送する。

3. 応答メッセージがクライアントの装置に届くと、それに対する確認通知メッセージをその装置のカーネルはサーバの装置のカーネルに返す。確認通知メッセージがタイムアウト時間内に受け取らなければ、サーバは応答メッセージを再送する。

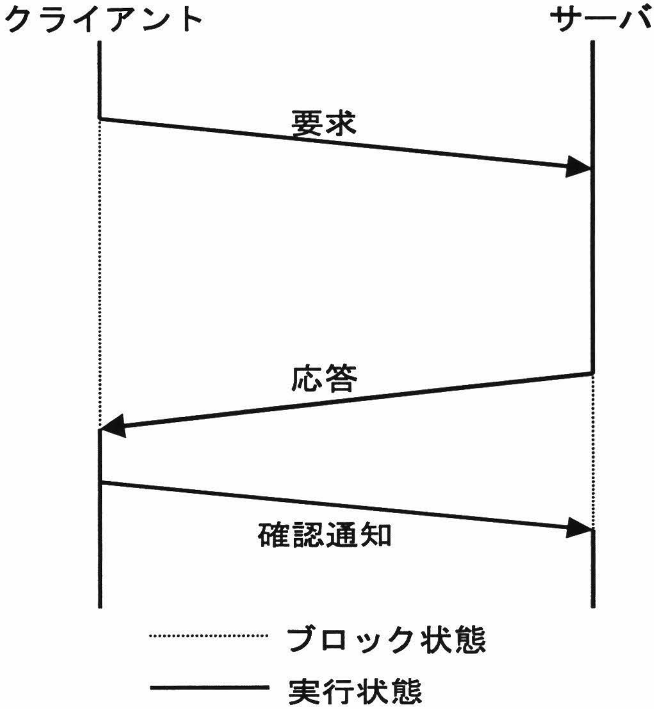


図 3.8　2 つのプロセス間でのクライアントサーバ通信の 3 メッセージ高信頼 IPC プロトコル

図 3.8 のプロトコルで、要求の処理が長くなると問題が起きる。要求メッセージが失われると、タイムアウト時間を経過した時のみ再送され、このタイムアウト時間は要求メッセージの不必要な再送を避けるために大きな値に設定される。一方、タイムアウト時間が要求の処理に十分なものでなければ、要求メッセージの不必要な再送を招く。次のようなプロトコルがこの問題に対処するために考えられている。

1. クライアントはサーバに要求メッセージを送る。

2. サーバの装置が要求メッセージを受けると、その装置のカーネルはタイマを起動する。タイムアウトする前にサーバがクライアントの要求の処理を完了し、クライアントに応答メッセージを返せば、応答は要求メッセージの確認通知ともなる。さもなければ、サーバ装置のカーネルによって要求メッセージの

応答として個別の確認通知メッセージが送られる。タイムアウト内に応答メッセージが受信されなければ、クライアントの装置のカーネルは要求メッセージを再送する。

3. 応答メッセージがクライアントの装置に届くと、それに対する確認通知メッセージをその装置のカーネルはサーバの装置のカーネルに返す。確認通知メッセージがタイムアウト時間内に受け取らなければ、サーバ装置のカーネルは応答メッセージを再送する。

図 3.8 のプロトコルで、クライアントからサーバ装置への確認通知メッセージは、便利であるが不必要である。これは、応答メッセージが紛失すると、タイムアウト後に要求メッセージが再送されるためである。サーバはもう一度要求を処理し、クライアントに応答を返す。したがって、メッセージパッシングシステムは、2 つのプロセス間のクライアントサーバ通信に対して次のような 2 メッセージ IPC プロトコルを使用するように設計され得る (図 3.9 参照)。

1. クライアントは要求メッセージをサーバに送り、サーバからの応答が返るまでブロックされる。

2. サーバがクライアントの要求の処理を完了すると、応答メッセージ (処理の結果を含む) をクライアントに返す。タイムアウト時間内に応答が受信できなければ、クライアント装置のカーネルは要求メッセージを再送する。

図 3.9 のプロトコルに基づいた場合の 2 つのプロセス間の通信中の障害処理の例を図 3.10 に示す。図 3.9 のプロトコルは、**最低 1 回セマンティクス (at-least-once semantics)** と言われ、少なくとも受信プロセスの 1 回の実行動作が行われることを保証している (可能ならば複数回)。この方法は、障害のために要求の実行が副作用を持っていたとしても要求に対する最後の実行の結果が送信側によって使用されるため、より適切に言えば**最後の 1 回セマンティクス (last-one semantics)** と呼べる。後で説明するように、この方法は幾つかのアプリケーションには適用できない。

3.9.1 重複要求メッセージの繰り返し等価と処理

繰り返し等価 (idempotency) とは、基本的に繰り返しの等価性を意味する。すなわち、繰り返す等価な動作は、同じ条件で何度行っても何らの副作用

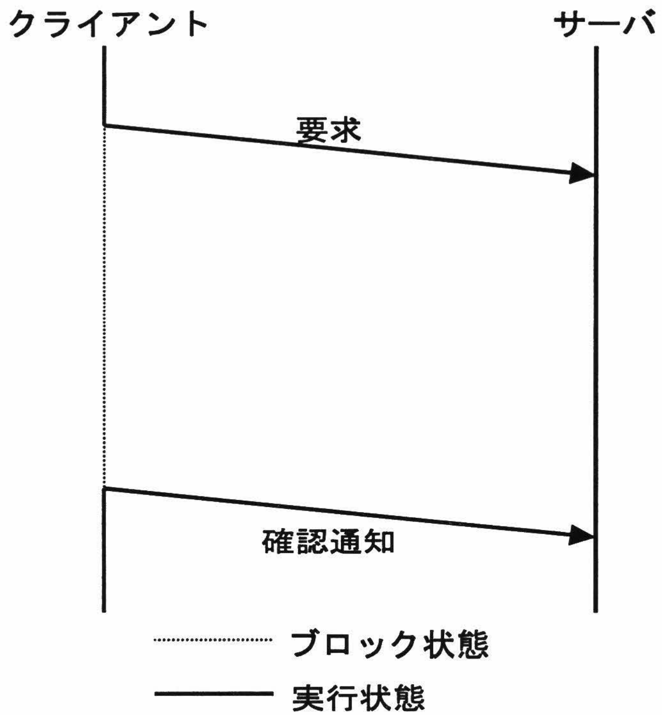


図 **3.9**　2 つのプロセス間でのクライアントサーバ通信の 2 メッセージ高信頼 IPC プロトコル

も生じなくて同じ結果をもたらすことである。繰り返し等価なルーチンの例は、与えられた数に対する平方根を計算する簡単な Getsqrt 手続きである。例えば、Getsqrt(64) は必ず 8 となる。

一方、同じ条件で繰り返し実行した場合に同じ結果を出さなくても良い動作を**非繰り返し等価 (nonidempotent)** であると言う。例えば、銀行の残高から指定された金額を引き去り、その結果を要求したクライアントに返すサーバプロセスの下記のルーチンを考える：

```
debit(arount)
if(balance≥ amount)
 {balance = balance - amount);
 return("success", balance);}
```

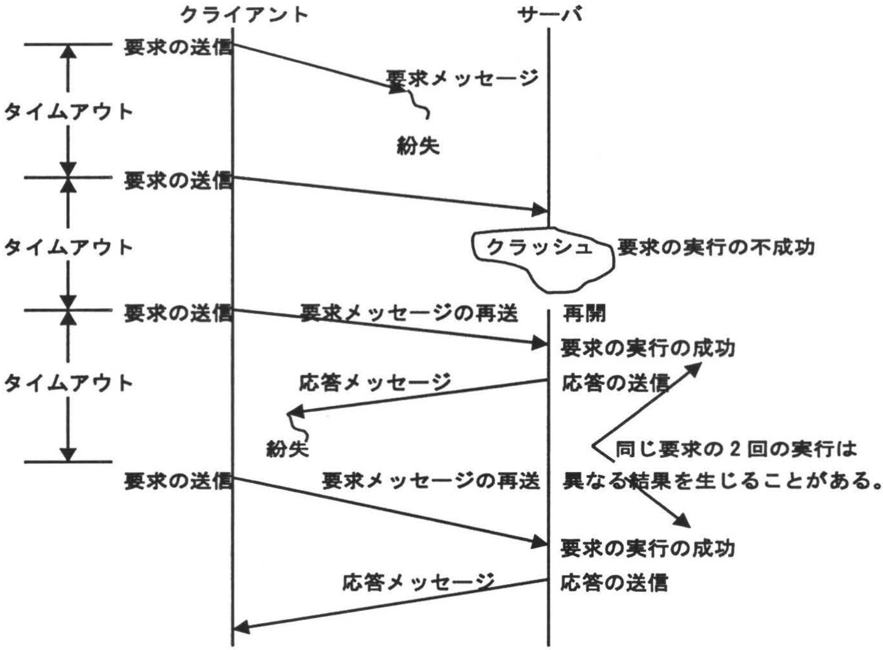
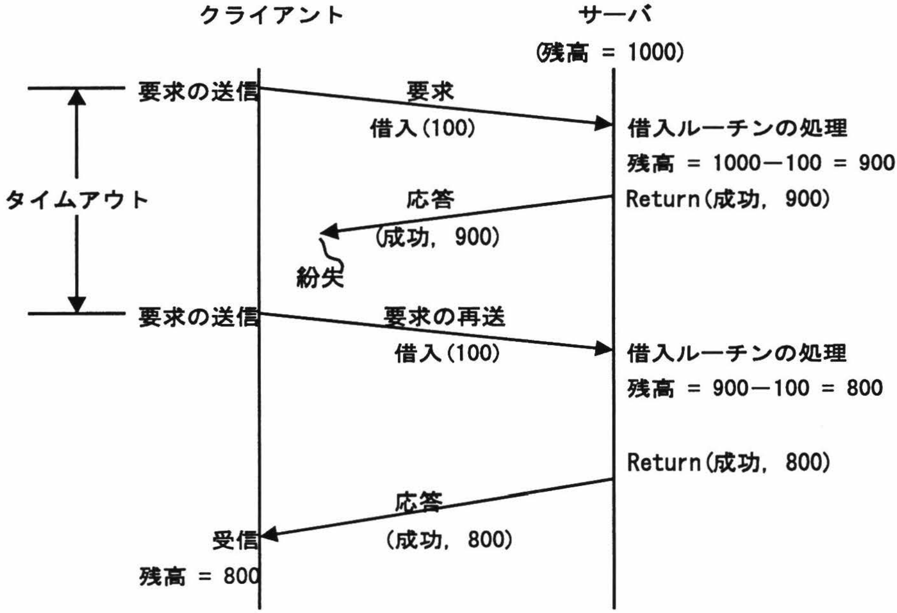


図 **3.10**　クライアントサーバ間のフォルトトレラント通信の例

else return(“failure”, balance);
end;

図 3.11 は、借入ルーチン処理でのクライアントからの借入 (100) 要求のシーケンスを示す。最初の要求は、残高から 100 を引き去ることをサーバに要求する。サーバはこの要求を受信し、処理を行う。初期の残高が 1000 とすると、サーバは残高として 900 があることを示すためにクライアントに応答 (成功, 900) を送る。何かの理由でこの応答がクライアントに届かないことがある。クライアントはこの要求の応答を待ち、タイムアウトになると、借入 (100) の要求を再送する。サーバは借入 (100) の要求を再度処理し、正しくはないが残高 800 を示すためにクライアントに応答 (成功, 800) を送る。したがって、非繰り返し等価のルーチンを複数実行すると期待しない結果をもたらすことがこの例から分かる。

明らかに、クライアントによって応答が受信されない場合、障害としてサー




```
借入(引き出し額)
{
    if(残高 >= 引き出し額)
    {  残高 = 残高 - 引き出し額;
       return ("成功", 残高);
     }
    else return("失敗"), 残高);
}
```

図 **3.11**　非繰り返し等価ルーチン

バの故障か要求又は応答メッセージの損失なのかを決めることができない。したがって、図 3.10 で分かるように、タイムアウトによる要求の再送を行うとサーバは同じ要求メッセージを何度も実行してしまう。この動作は、要求の複数の実行が 1 回の実行と同じ結果を持つかどうかに (繰り返し等価なルーチン) よってよい場合もあれば悪い場合もある。もし、要求の実行が非繰り返し等価であれば、その繰り返された実行によって情報の一貫性を壊してしまう。したがって、このような受け入れられない実行は、一般に避けねばならない。受け入れられない現象は識別され、サーバでの実行がただ 1 回しか起きないように保証するためにちょうど 1 回セマンティクス (**exactly-once semantics**) を使用しなければならない。ちょうど 1 回セマンティクスに基づくプリミティブは、望まれるところであるが実現が困難である。

ちょうど 1 回セマンティクスの実現方法の 1 つとして、クライアントの全ての要求にユニークな識別子を与え、応答を一時的に保持するためにサーバ装置のカーネルのアドレス空間に応答キャッシュを設定することである。この場合、処理のためにサーバに要求を転送する前に、サーバ装置のカーネルは要求に対して既に応答キャッシュに応答が存在するかを確認する。もしあれば、既に処理しており重複した要求となる。したがって、以前の計算結果を応答キャッシュから取り出し、クライアントに応答する。ない場合には新しい要求である。この場合、カーネルは処理のためにサーバに要求を渡し、処理が完了すると、クライアントに応答メッセージを送る前に処理結果とともに要求識別子を応答キャッシュに保持する。

ちょうど 1 回セマンティクスの実現例を図 3.12 に示す。この図は、要求に番号があることを除けば図 3.11 と同じであり、応答キャッシュがサーバ装置に付加されている。クライアントは要求 1 を作成する。サーバ装置のカーネルは要求 1 を受信し、要求 1 の応答が応答キャッシュにあるかを調べる。なければ、要求をサーバに渡す。サーバは要求を処理し、カーネルに結果を返す。カーネルは要求識別子と実行結果を応答キャッシュにコピーし、クライアントにその結果を送信する。この応答が紛失し、クライアントでタイムアウトが発生すると要求 1 が再送される。サーバ装置のカーネルは、再び要求 1 を受信し、要求 1 の応答が応答キャッシュにあるかを調べる。この場合、見つかるため応答キャッシュから要求 1 に対応する応答を取り出し、再度応答メッセージをクライアントに送信する。要求識別子はキャッシュの登録数よりも大きくする必要がある。

応答キャッシュの使用は非繰り返し等価なルーチンを繰り返し等価なものにするものではない点に注意することは重要である。キャッシュは、単に非繰り

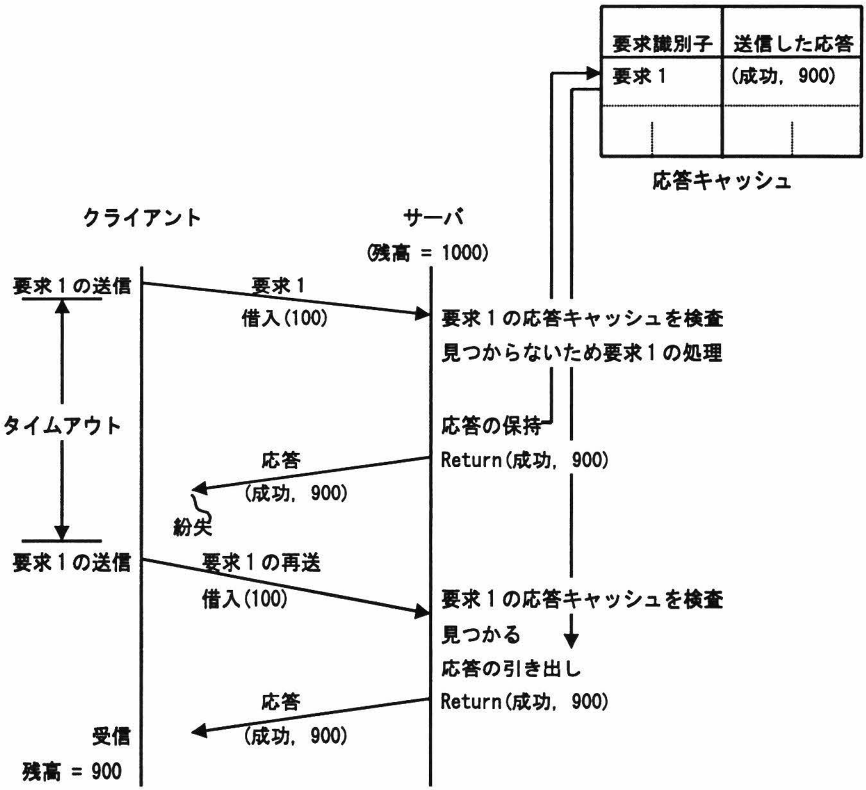


図 **3.12**　要求識別子と応答キャッシュを用いたちょうど 1 回セマンティクスの例

返し等価なルーチンをちょうど1回セマンティクスに実現する1つの方法にすぎない。

3.9.2 マルチデータグラムメッセージの紛失と順序誤りパケットの回復

マルチデータグラムメッセージの場合、メッセージの転送は、幾つかのパケット転送によって行われる。したがって、メッセージの転送は、メッセージの全てのパケットが宛先のプロセスによって受信された時にのみ完了したと考えられる。マルチデータグラムメッセージが正しく完了するためには、全てのパケットについて信頼性のある送達が重要である。これを保証する簡単な方法は、各パケット毎に確認通知を持つことである(**逐次応答プロトコル(stop-and-wait protocol)** と呼ばれる)。各要求パケットに対する確認通知は通信オーバヘッドを招く。したがって、通信性能を改善するために、マルチデータグラムメッセージの全てのパケットに対して1つの確認通知パケットを使用する方法がある(**一括応答プロトコル (blast protocol)** と呼ばれる)。しかしながら、この方法でノードのクラッシュや通信リンクの障害が発生すると次のような問題を引き起こす。

■マルチデータグラムメッセージのパケットが通信中に紛失する。
■受信側にパケットが正しい順番で受信されない。

この問題に対する有効な機構は、メッセージのパケットを識別するためのビットマップを使用することである。この機構では、各パケットのヘッダ部分に2つの新たなフィールドを持ち、1つはマルチデータグラムメッセージを構成するパケット数を示し、他はメッセージ内のパケットの位置を示すビットマップである。最初のフィールドは受信プロセスがメッセージを受信するための適切なバッファ領域を用意するのに有効であり、2番目はバッファにおけるパケットの位置を決めるのに役立つ。全てのパケットがメッセージを構成するパケットの数の情報を持っているため、パケットの順番がくるったとしても、すなわち最初のパケットが最初に届かなかった場合でも、必要なバッファ領域は受信プロセスによって準備され、受信されたパケットはバッファ内に適切に配置される。タイムアウトが発生した時、全てのパケットが受信されていなければ、受信していないパケットを示すビットマップを送信プロセスに返す。こ

のビットマップ情報によって受信されていないパケットを送信プロセスは再送する。この技術は、選択再送 **(selective repeat)** と呼ばれる。マルチデータグラムメッセージの全てのパケットが受信されると、メッセージの転送が完了し、受信プロセスは送信プロセスに確認通知メッセージを返す。このマルチデータグラムメッセージ通信の方式の例として 5 つのパケットから成るシーケンスを図 3.13 に示す。

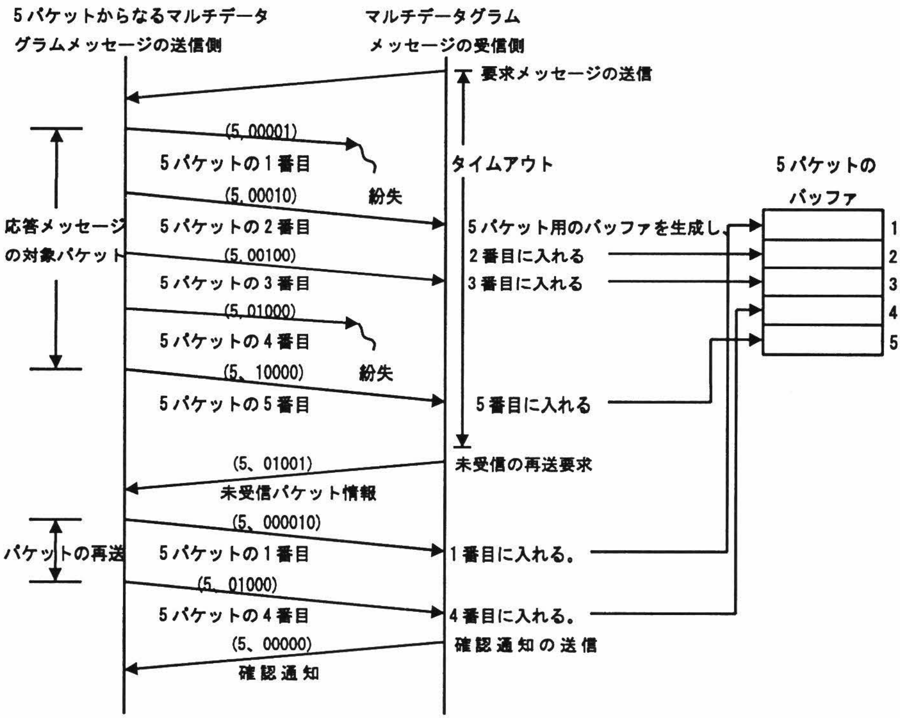


図 3.13　マルチデータグラムメッセージ転送における順序誤りパケットの回復を行うためのビットマップの使用例

3.10　グループ通信

メッセージの交信におけるもっとも基本的な形態は、**1 対 1 通信 (one-to-one communication)** であり (**2 地点間 (point-to-point)**、**ユニキャスト (unicast)** 又は単に**通信 (communication)** としても呼ばれる)、単一の送信

プロセスが単一の受信プロセスにメッセージを送信する。しかしながら、プログラミングの性能や容易性のために、幾つかの高度に並列処理を行う分散アプリケーションは、メッセージパッシングシステムはグループ通信機能を提供することを要求する。送信及び受信が単一又は複数かによってグループ通信は次の3つに分類できる。

1. 1対多 (1つの送信プロセスと複数の受信プロセス)
2. 多対1(複数の送信プロセスと1つの受信プロセス)
3. 多対多 (複数の送信プロセスと複数の受信プロセス)

これらの方式に関する課題について次に説明する。

3.10.1 1対多通信

この方式では、1つの送信プロセスが送信するメッセージに対して複数の受信プロセスがいる。1対多の方式は、**マルチキャスト通信 (multicast communication)** と呼ばれる。マルチキャスト通信の特殊な場合が**ブロードキャスト通信 (broadcast communication)** であり、メッセージはネットワークに接続されている全ての装置に送信される。

マルチキャスト及びブロードキャスト通信は、幾つかの実際的なアプリケーションに大変有効である。例えば、全て同じタイプのサービスを提供するサーバグループの管理を行うサーバ管理者を考える。このサーバ管理者は、全てのサーバプロセスにメッセージをマルチキャストし、要求を処理できるサーバを募る。これに応答した最初のサーバが選ばれる。サーバ管理者は空きのサーバを持っておく必要がない。同様に、特殊なサービスを提供する装置を割り当てるために、問い合わせメッセージがブロードキャストされるかもしれない。この場合、全ての装置から回答を受信する必要はなく、要求するサービスの1つを見つけるだけで十分である。

グループ管理

1対多通信においては、メッセージの受信プロセスはグループを形成する。このようなグループには2つのタイプ、閉域と開放がある。**閉域グループ (closed group)** はグループのメンバー内にのみメッセージを送ることができる。グループ外のプロセスはグループの個々のメンバーにメッセージを送るこ

とはできるが、グループ全体としてメッセージを送ることはできない。一方、**開放グループ (open group)** はシステムのプロセスはグループ全体としてメッセージを送ることができる。

閉域グループ又は開放グループのどちらを使用するかはアプリケーションに依存する。例えば、共通の問題に関係するグループはそのグループ外のプロセスとは通信する必要がなく、閉域グループを構成できる。一方、クライアントの要求を分散処理として行う複数のサーバからなるグループは、クライアントプロセスはサーバに要求を送ることができるために、開放グループを構成しなければならない。したがって、グループ通信機能を持つ柔軟性のあるメッセージパッシングシステムは 2 つのタイプのグループに対応すべきである。

グループ通信機能を持つメッセージパッシングシステムは、動的にグループを生成や削除を行い、自由にプロセスがグループに加入したり、脱退することができるような柔軟性を提供する。したがって、メッセージパッシングシステムはグループ及びそのメンバー構成情報を管理するための機構を持たなければならない。このための簡単の機構は、集中型の**グループサーバ (group server)** のプロセスを使用することである。グループの生成、削除、グループへの加入、脱退の全ての要求は、このプロセスに送られる。したがって、グループサーバは散在するグループやそのメンバー構成の情報を更新管理することは容易である。しかしながら、この方法はす全て集中化技術によるために信頼性と規模拡張性の共通的な問題がある。この問題の解決の 1 つとしてグループサーバの多重化がある。しかしながら、この方法はグループサーバ間の情報の一貫性を確保するためのオーバヘッドをもたらす。

グループアドレス付け

一般に 2 レベルの名前方式がグループアドレス付けに使用される。高位レベルグループ名は、グループのプロセスの位置情報に依存しない ASCII 文字列である。一方、低位レベルグループ名は使用するハードウェアに強く依存する。例えば、あるネットワークにおいて、複数の装置が意識する特別なネットワークアドレスを生成することができる。このようなネットワークアドレスは、**マルチキャストアドレス (multicast address)** と呼ばれる。マルチキャストアドレスで送られるパケットは、このアドレスを関わる全ての装置に自動的に転送される。したがって、このようなシステムではマルチキャストアドレスはグループに対して低位レベル名として使用される。

マルチキャストアドレスを生成する機能を持たないネットワークは、ブロー

ドキャスト転送機能を持つことがある。ブロードキャスト転送機能を持つネットワークはあるアドレス、例えば**ブロードキャストアドレス (broadcast address)** として値ゼロを宣言する。したがって、ブロードキャストアドレスは、グループに対して低位レベル名として使用される。この場合、各装置のソフトウェアはパケットがこのアドレスかを調べなければならない。このアドレスでなければ、パケットは単に廃棄される。全ての装置が全てのブロードキャストパケットを受信し、パケットを調べなければならないために、ブロードキャストアドレスの使用はグループアドレス付けのためにマルチキャストアドレスの使用よりも効果的でない。また、グループアドレス付けのためにブロードキャストアドレスを使用するシステムでは、全てのグループはブロードキャストアドレスとして同じ低位レベル名を持つ。

ネットワークがマルチキャストアドレスやブロードキャストアドレス機能を持っていない場合、1 対 1 通信機構がグループ通信機能を実現するために使用されなければならない。すなわち、送信装置のカーネルはグループに所属するプロセスの各装置に個別にメッセージパケットを送信するために使用される。したがって、この場合、グループの低位レベル名はグループに属するプロセスの装置識別子のリストを含む。

最初の 2 つの方式では 1 つのメッセージパケットがネットワークに転送され、3 番目の方式ではネットワークに転送されるパケット数はグループに所属するプロセスの装置の数に依存する。したがって、3 番目の方式は他の 2 つの方式に比べて多くのネットワークトラヒックを生成するため、一般的に効果的でない。しかしながら、ほとんどのグループが少ない装置から構成されている場合にはブロードキャスト方式よりも有効である。さらに、最初の 2 つの方法は単一の LAN での使用に適している。ネットワークがゲートウェアによって相互接続された複数の LAN を含む場合、3 番目の方法は他の 2 つの方法よりも実現することが単純であり、容易である。

受信プロセスへのメッセージ転送

ユーザアプリケーションは、プログラムで高位レベルグループ名を使用する。集中型のグループサーバは高位レベルグループ名と低位レベルグループ名の対応を管理する。グループサーバは、また各グループのプロセスの全てのプロセス識別子のリストを管理する。

送信プロセスが高位レベル名で指定されたグループにメッセージを送信した時、その装置のカーネルはグループの低位レベル名とグループに属するプロ

セスのプロセス識別子のリストを得るためにグループサーバに問い合わせる。プロセス識別子のリストはメッセージパケットに入れられる。もし、低位レベルグループ名がマルチキャストアドレスやブロードキャストアドレスの場合、カーネルは単にマルチキャスト又はブロードキャストアドレスでパケットを送信する。一方、低位レベルグループ名が装置の識別子のリストであれば、カーネルはこのリストの各装置にそれぞれ同じパケットを送信する。

パケットが装置に届くと、その装置のカーネルはパケットからプロセス識別子のリストを確認し、その装置に属するリストの中のプロセスにパケットの中のメッセージを転送する。ブロードキャストアドレスが低位レベルグループ名として使用された場合、受信した装置に属するプロセスがいないこともある。この場合、カーネルはそのパケットを廃棄する。

送信プロセスは、グループの大きさやグループアドレス付けに使用する実際の機構については全く関知しない。送信プロセスは、単に高位レベルグループ名で指定されたグループにメッセージを送り、オペレーティングシステムがグループメンバーの全てにメッセージを転送する責任を持つ。

バッファ方式と非バッファ方式によるマルチキャスト

マルチキャスト転送は、非同期通信機構である。これは、マルチキャストの送信が次の理由から同期ではできないためである [Gehani 1884]。

1. マルチキャストグループに属する全ての受信プロセスがマルチキャストメッセージが受信できる状態まで送信プロセスが待つことは非現実的である。
2. 送信プロセスは、マルチキャストグループに属する全ての受信プロセスを意識しない。

マルチキャストメッセージが受信プロセス側でどのように扱われるかは、マルチキャスト機構がバッファ方式か非バッファ方式かに依存する。非バッファ方式によるマルチキャストでは、メッセージは受信プロセスでバッファに置かれなく、もし受信プロセスが受信する用意ができていなければメッセージは失われる。一方、バッファ方式によるマルチキャストでは、メッセージは受信プロセスでバッファに置かれ、マルチキャストグループの各プロセスはそれぞれの状態にあわせてメッセージを受信する。

全送信と掲示板セマンティクス

Ahamad と Bernstein[1985] は、1 対多通信に対して次の 2 つのセマンティクスのタイプを記述している。

1. **全送信セマンティクス (send-to-all semantics)**:メッセージのコピーがマルチキャストグループの各プロセスに送られ、メッセージはプロセスが受信するまでバッファに置かれる。

2. **掲示板セマンティクス (bulletin-board semantics)**:マルチキャストされるメッセージはマルチキャストグループの全ての個々のプロセスに送られる代わりにチャネルに送られる。論理的な見方をすれば、このチャネルは掲示板の役目を果たす。チャネル上に受信要求が発生するとそれを取り去るのではなく、チャネルからメッセージをコピーする。このように、マルチキャストメッセージは掲示板に置かれたように他のプロセスにも受け取れる。チャネル上の受信アクセス権を持つプロセスはマルチキャストグループを構成する。

掲示板セマンティクスは、全てへの送信セマンティクスよって扱われていない次の 2 つの点が考慮されているためより柔軟性がある [Ahamad and Bernstein 1985]。

1. 特定の受信プロセスに関連するメッセージは受信プロセスの状態に依存する。
2. 転送後ある時間内に受信できないメッセージはもはや意味がないかもしれない。この価値は送信プロセスに依存する。

このことを説明するために、もう一度要求をサービスできるサーバ装置を見つけるためにサーバ管理者がメッセージをマルチキャストする例を考える。全てへの送信セマンティクスでは、全てのサーバにマルチキャストすることが必要になり、多くの関係プロセスが処理を行うことになる。掲示板セマンティクスでは、空いていて要求をサービスできるものだけがチャネル上の受信要求を受け、その時点で可能なもののみがこのメッセージを処理する [Ahamad and Bernstein 1985]。さらに、メッセージは決められた時間が経過するとサーバ管理者によってチャネルから除かれる。すなわち、最初に決めたものが選ばれる。したがって、サーバ管理者が提案されたものを受け入れられる状態にいる限りメッセージは受信できる状態に置かれる。これは完全に余分なメッセージを押

さえることができないが(ある時間を過ぎても応答を返すかもしれない)、減少することはできる。

マルチキャストの柔軟性のある信頼性

アプリケーションは、いろいろな信頼性の程度を要求する。したがって、マルチキャストのプリミティブは柔軟性のあるユーザ定義可能な信頼性を提供する。マルチキャストの送信プロセスは、応答メッセージが期待できる受信プロセスを指定することができる。1対多通信では、信頼性の程度は、次のような形式で表現される。

1. **0信頼性 (0-reliable)**:送信プロセスは、いかなる受信プロセスからの応答を期待できない。これは、送信プロセスがマルチキャストメッセージを送信した後応答を待たない非同期のマルチキャストを使用するアプリケーションに有効である。このようなアプリケーションの例として時刻信号生成がある。

2. **1信頼性 (1-reliable)**:送信プロセスは受信プロセスから1つの応答を期待する。サーバ管理者が要求をサービスできるサーバを見つけるためにメッセージをマルチキャストし、応答した最初のサーバを選択するアプリケーションは、この1信頼性マルチキャスト通信の例である。

3. **n中m信頼性 (m-out-of-n-reliable)**:マルチキャストグループは、nの受信プロセスから構成され、送信プロセスはその中の$m(1 < m < n)$の受信プロセスからの応答を期待する。重複情報の一貫性制御に使用される多数決アルゴリズム(第9章で説明)は、$m = n/2$の値の場合の信頼性の形式を使用する。

4. **全信頼性 (all-reliable)**:送信プロセスは、マルチキャストグループの全ての受信プロセスからの応答を期待する。例えば、多重ファイルの更新のためのメッセージはそのファイルを持っている全てのファイルサーバにマルチキャストされる。このような場合には、送信プロセスは関係する全てのファイルサーバからの応答を期待する。

原子マルチキャスト

原子マルチキャストは、すべてか無の特性を持つ。すなわち、メッセージが原子マルチキャストによってグループに送られた場合、グループの全てのメン

バーが受信するか若しくは誰も受信しないどちらかである。原子マルチキャストにおける暗黙の前提は、プロセスが障害となるともはやグループのメンバーではない。プロセスが復旧すると、改めてグループに加入しなければならない。

原子マルチキャストは、必ずしも必要ではない。例えば、信頼性の程度として0信頼性、1信頼性又はn中m信頼性を要求するアプリケーションは、原子マルチキャスト機能を必要ではない。一方、信頼性の程度として全信頼性を要求するアプリケーションは、原子マルチキャスト機能を必要とする。したがって、柔軟性のあるメッセージパッシングシステムは原子及び非原子マルチキャスト機能を持つべきであり、マルチキャストされるメッセージに原子性が要求されるかどうかを送信プリミティブに指定することでマルチキャストメッセージの送信プロセスに柔軟性を提供すべきである。

原子マルチキャスト実現する簡単な方法として、全信頼性の要求を持つメッセージをマルチキャストすることである。この場合、送信プロセスの装置のカーネルは、グループの全てのメンバーにメッセージを送信し、各メンバーからの確認通知を待つ(1対1通信機構がマルチキャスト機能を実現することを仮定している)。タイムアウトになると、カーネルは確認通知メッセージを受信していないメンバーに対してメッセージを再送する。タイムアウトに基づくメッセージの再送は、グループの全てのメンバーから確認通知を受けるまで繰り返す。全ての確認通知を受けると、カーネルは原子マルチキャスト処理が完了したことを送信プロセスに通知する。

上記の動作は、送信プロセスと受信プロセスが原子マルチキャスト動作中に障害にならない時に限りうまくいくものである。送信プロセスが障害になれば、パケットの紛失や他の原因でメンバーがメッセージを受け取れない場合にメッセージの再送ができなくなるためである。同様に、受信プロセスの装置が障害で動作しなくなった場合、メッセージの再送は無限に続くことになり、ある時間を経過すると止めることになるためにメッセージは転送されない。したがって、フォールトトレラント性の原子マルチキャストプロトコルは、マルチキャストが送信側の装置や受信側の装置が障害になった場合でさえもマルチキャストグループの全てのメンバーに送達されることを保証しなければならない。このようなプロトコルの1つの実現方法を次に述べる [Tanenbaum 1995]。

この方法では、各メッセージは他のメッセージと区別するためにメッセージ識別子フィールドと原子マルチキャストを示すフィールドを持つ。送信プロセスはメッセージをマルチキャストグループに送信する。送信側の装置のカーネルは、このメッセージをマルチキャストグループの全てのメンバーに送信し、

既に述べた方法のタイムアウトに基づくメッセージ再送方法を使用する。メッセージを受信するプロセスは、新しいメッセージかをメッセージ識別子フィールドによって調べる。新しいメッセージでなければ廃棄する。そうでなければ、原子マルチキャストメッセージかを調べる。原子マルチキャストメッセージであれば、そのメッセージをグループメンバーに転送するためにその原子マルチキャストを遂行する。この装置のカーネルは、このメッセージを通常の原子マルチキャストメッセージとして扱い、必要な時にタイムアウトの再送を使用する。この方法では、原子メッセージキャストメッセージの受信プロセスは、同じマルチキャストグループにこのメッセージの原子マルチキャストを遂行する。この方法は、それぞれ、動作しているマルチキャストグループプロセスは、メッセージを送信した後の送信プロセスの装置の障害やメッセージ受信後の受信プロセスの装置の障害があっても、メッセージを受信する。

原子マルチキャストは、一般に実現する上では数多くのメッセージの含むため通常のマルチキャストに比べて重装備である。したがって、メッセージパッシングシステムはこの原子性を既定値として使用しないで、オプションとして使用すべきである。

グループ通信プリミティブ

1 対 1 通信と 1 対多通信とも、送信プロセスは基本的に 2 つのパラメータ、宛先アドレスとメッセージだデータへのポインタを指定する。したがって、理想的には 1 対 1 通信と 1 対多通信に対して同じ送信プリミティブが使用できる。送信プリミティブの宛先アドレスが単一プロセスであれば、メッセージは 1 対 1 通信である。一方、宛先アドレスがグループアドレスであればメッセージはグループに属する全てのプロセスに送信される。

しかしながら、グループ通信機能を持つ多くのシステムは、メッセージをグループに送信するために異なるプリミティブ (例えば、send_group) を提供する。これには大きく 2 つの理由がある。1 つは、グループ通信機能の設計と実現が簡単にできる。例えば、プロセスアドレス付けとグループアドレス付けに 2 レベルの名前機構が使用されているとすると、プロセスに対する高位レベルから低位レベルの対応は名前サーバによって行われ、グループに対する対応はグループサーバによって行われる。この設計では、1 つの送信プリミティブが 1 対 1 通信と 1 対多通信に使用される場合、送信側のカーネルはユーザが指定した宛先アドレスが単一プロセスアドレスかグループアドレスかが理解できない。したがって、低位レベル名を得るために名前サーバ又はグループサーバの

どちらに問い合わせれば良いかがわからない。この問題を解決する実現方法は、論理的には可能であるが、設計は複雑になる。一方、別々のプリミティブ、例えば send と send_group を使用すると、カーネルは指定された宛先アドレスが単一プロセスアドレスかグループアドレスかが簡単に分かり、低位レベル名を得るための適切なサーバに問い合わせることができる。

2つ目として、ユーザにより柔軟性を提供することができる。例えば、ユーザが信頼性の程度 (応答を期待する受信プロセス数) を指定するためのパラメータを send_group に追加でき、また別のパラメータとして原子性が要求されているか否かを指定できる。

3.10.2 多対1通信

この方式では、複数の送信プロセスが1つの受信プロセスにメッセージを送る。単一の受信プロセスは選択又は非選択的である。**選択的受信者 (selective receiver)** は特定の送信プロセスを指定する。メッセージの交信は、その送信プロセスがメッセージを送信した時のみ行われる。一方、**非選択的受信者 (nonselective receiver)** は送信プロセスの集合を指定し、その集合の中の1プロセスからこの受信プロセスにメッセージが送られるとメッセージの交信が行われる。

このように、多対1通信方式に関連する重要な問題は非決定的である。受信プロセスは、送信グループの特定のプロセスからよりもむしろそのグループの中からの情報を待つかもしれない。グループの誰が最初にその情報を用意できるかは分かるないように、このような振る舞いは非決定的であると言える。ある場合では、グループのどの送信プロセスからメッセージを受けるかを動的に制御することは有効である。例えば、バッファプロセスはバッファが一杯にならない時には何時でもバッファに格納するために生産者プロセスからの要求を受理する。一方、バッファプロセスは、バッファが空きでなければ何時でもバッファから情報を獲得するための消費者プロセスの要求を受け入れる。このような振る舞いをプログラム化するためには、非決定処理を表現し、制御するための記述法は必要である。1つの方法として Dijkstra[1975] によって紹介された「ガードコマンド」文がある。このような問題はオペレーティングシステムよりもむしろプログラミング言語に関連するため、ここではこれ以上言及しない。

3.10.3　多対多通信

この方式では、複数の送信プロセスが複数のにメッセージを送信する。1 対多通信や多対 1 通信はこの方式に含まれる。したがって、既に述べて 1 対多通信や多対 1 通信に関する課題は、多対多通信にも関係する。さらに、この多対多通信方式に関係する重要な課題として**順序メッセージ配送 (ordered message delivery)** がある。

順序メッセージ配送は、全てのメッセージがアプリケーションに受信可能な順序で全ての受信プロセスに配送されることを保証する。この特性は、多くのアプリケーションにとって正しい機能を行うためには必要である。例えば、データベースの複製を持つ 2 つのサーバプロセスにデータベースの同じレコード更新するために、2 つの送信プロセスがメッセージを送信することを考える。2 つの送信プロセスのメッセージが 2 つのサーバに異なる順序で届くと、データベースの更新されたレコードの最終的な値はそれぞれのデータベースでは異なる。したがって、このようなアプリケーションは全てのメッセージが全ての受信プロセスに同じ順序で配送されることを要求する。順序メッセージ配送はメッセージ順序番号を要求する。単一送信プロセスで複数受信プロセス (1 対多通信) のシステムでは、全ての受信プロセスへのメッセージの順序番号は一般的である。送信プロセスがマルチキャストメッセージが全ての受信プロセスに受信されたことを確認した時のみ次のマルチキャストメッセージの送信に移るならば、メッセージは同じ順序で配送される。一方、複数の送信プロセスで単一の受信プロセス (多対 1 通信) のシステムでは、メッセージは受信側の装置に到着した順序で受信プロセスに配送される。この場合の順序性は受信プロセスによって処理される。このように、1 対多通信や多対 1 通信方式ではメッセージの順序性を保証することは困難ではない。

しかしながら、多対多通信では、ある送信プロセスから送信されたメッセージは他の送信プロセスからのメッセージよりも前に受信プロセスに到着するかもしれない。しかし、この順序は他の受信プロセスでは逆になるかもしれない (図 3.14 参照)。異なる送信プロセスからのメッセージが別の受信プロセスに異なる順序で到着する理由として、2 つのプロセスが LAN でのアクセスの競合によるようなものであり、LAN 上での複数のプロセスからのメッセージの順序は非決定的である。さらに、WAN の環境では、異なる送信プロセスのメッセージは送信時間が異なる別の経路で同じ宛先に届くことがある。したがって、順序メッセージ配送を保証するためには多対多通信方式において特別

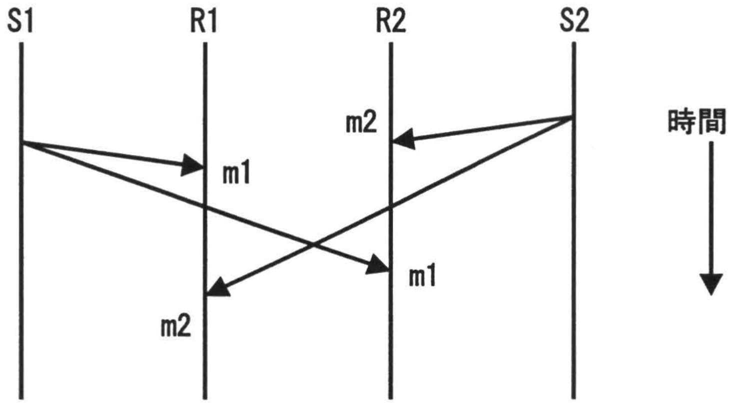


図 **3.14** メッセージ配信での順序の非一貫性

なメッセージ処理機構が必要となる。

順序マルチキャストメッセージの配送のための共通的に使用される方式は、**絶対順序性 (absolute ordering)**、**一貫順序性 (consistent ordering)**、**因果順序性 (causal ordering)** である。

絶対順序性

この方式は、全てのメッセージが送られた順序で全ての受信プロセスに配送されることを保証する (図 3.15 参照)。この方式を実現する 1 つの方法として、メッセージ識別子として全体で分かるタイムスタンプの使用がある。すなわち、システムの各装置は時計を持ち、全ての時計はお互いに同期し、メッセージを送信する時に時刻 (タイムスタンプ) をそのメッセージの識別子としてメッセージに挿入する。

各受信側の装置のカーネルは、入力する全てのメッセージを受信プロセス対応の個別のキューに入れる。スライディングウィンドウ機構がこのメッセージをキューから受信プロセスに定期的に配送するために使用される。すなわち、一定の時間間隔がウィンドウサイズとして選択され、その時点のウィンドウ内にあるタイムスタンプ値を持つ全てのメッセージが周期的に受信プロセスに配送される。タイムスタンプ値がウィンドウ内にないメッセージは、それらのメッセージよりも前のタイムスタンプ値を持つ遅れているメッセージが届くことがあるためキューに置かれる。ウィンドウサイズは、ある装置から他の装置

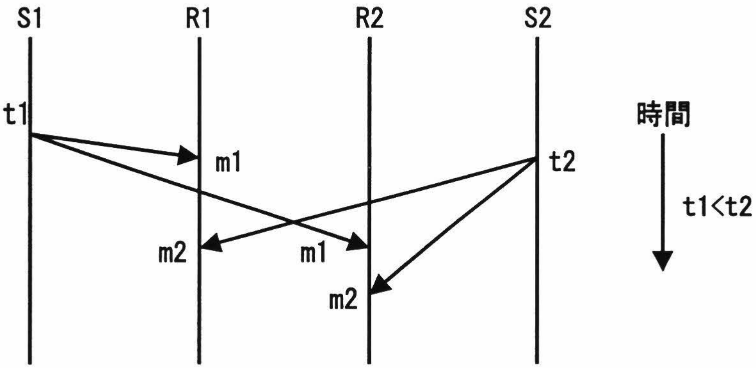


図 **3.15**　メッセージの絶対順序性

にメッセージがネットワークを転送するのに必要となる最大可能時間を考慮して適切に選択される。

一貫順序性

絶対順序性は、全体に同期させた時計が必要であり、実現することは容易ではない。さらに、この方式を現在の多くのアプリケーションが必要としている訳ではない。例えば、二重化したデータベースの更新の例で、2 つのサーバが 2 つの送信プロセスから更新メッセージを受信する場合、2 つのメッセージが送信された実際の順序でないとしても 2 つのサーバに届く順序が同じであることを保証できれば十分である。したがって、多くのシステムは、絶対順序性に代わってこのような一貫順序性を採用する。この方式は、全てのメッセージが全ての受信プロセスに同じ順序で配送されることを保証する。この順序は、実際にメッセージが送信された順序とは異なるかもしれない (図 3.16 参照)。

この一貫順序性を実現する 1 つの方法は、多対多通信を多対 1 通信と 1 対多通信の組み合わせとして実現することである [Chang and Maxemchuk 1985]。すなわち、送信側の装置のカーネルは、各メッセージに順序番号を割り当てかつそのメッセージをマルチキャストする単一の受信プロセス (**シーケンサ (sequencer)** と呼ばれる) にメッセージを送信する。各受信側の装置のカーネルは、入力する全てのメッセージを受信プロセス対応毎のキューに入れる。キューにあるメッセージはメッセージ識別子に順序性が保たれている限り、即座に受信プロセスに配送され、順序性が保たれていない時は順序が正しくなる

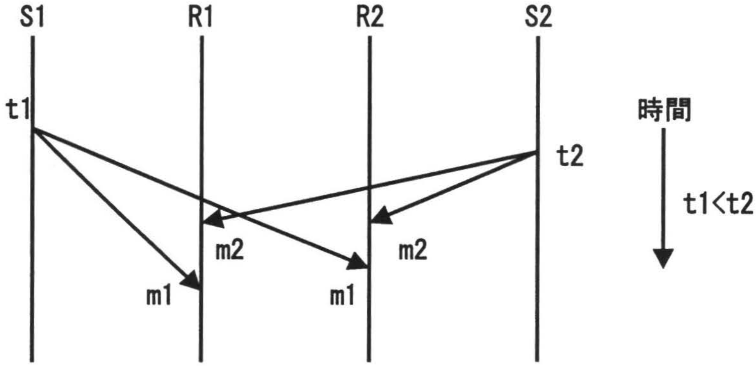


図 **3.16** メッセージの一貫性ある順序性

ようなメッセージを受信するまでキューに置かれる。

一貫順序性を実現するためのシーケンサに基づく方式は、一個所の障害に影響を受けるため、信頼度が弱い。この問題に対応するための一貫順序性を実現するための分散アルゴリズムは、ISIS システムの ABCAST プロトコルである [Birman and Van Renesse 1994, Birman 1993, Birman et al. 1991, Birman and Joseph 1987]。これは、グループメンバー内での分散合意によりメッセージに順序番号を割り当て、次のように動作する。

1. 送信プロセスは一時的な順序番号をメッセージに割り当て、マルチキャストグループの全てのメンバーに送信する。送信プロセスによって割り当てられた順序番号は、以前の値より大きい値でなければならない。したがって、メッセージに順序番号を割り当てるために送信プロセスは簡単なカウンタを使用する。

2. メッセージを受信すると、グループの各メンバーは送信プロセスに提案する順序番号を返す。メンバー (i) はこの提案する順序番号を下記の関数で計算する。

$$max(F_{max}, P_{max}) + 1 + i/N$$

ここで、Ｆｍａｘはグループによって受信されたメッセージに対して合意さ

れた最終最大順序番号 (各メンバーは最終順序番号が合意された時この値を記録する)、Ｐｍａｘはメンバーからの提案された最大順序番号、Nはマルチキャストグループのメンバーの数である。

3. 送信プロセスは全てのメンバーから提案された順序番号を受信すると、メッセージの最終順序番号として最大値をもつ順序番号を選び、commit メッセージで全てのメンバーに通知する。最終順序番号は、提案する順序番号の計算に使用される関数の i/N によって唯一性が保証される。

4. commit メッセージを受信すると、各メンバーはメッセージに最終順序番号を付与する。

5. 最終順序番号を持つ commit メッセージは、最終順序番号の順序でアプリケーションに配送される。メッセージの順序番号割り当てアルゴリズムは実行時のシステム機能の1部であり、ユーザプロセスの機能ではない。

このプロトコルは、一貫順序性を実現するものである。

因果順序性

幾つかのアプリケーションにとって、一貫順序性は必要ではなく、もっと弱い順序性でも良い。したがって、アプリケーションは、使用するメッセージパッシングシステムが受け入れ可能なも少し弱い順序性を提供するのであれば、より性能を向上できる。多くのアプリケーションが受け入れ可能な1つの順序性としては、因果順序性である。これは、1つのメッセージ送信の事象が、他のメッセージ送信の事象に因果関係があったとしても2つのメッセージは正しい順序で全ての受信プロセスに配送されることを保証する。しかし、2つのメッセージ送信の事象が因果関係がなければ、2つのメッセージは受信プロセスに異なる順序で配送されてよい。2つのメッセージ送信の事象は、それらが**先行生起 happened-before)** 関係 (先行生起関係の定義は第6章を参照) にあるならば因果関係にあると言われる。すなわち、2つのメッセージ送信の事象は、最初の事象に影響されて2番目の事象が発生した可能性があれば、因果関係を持つ。この因果順序性の基本的な考え方は、関係があればメッセージを適切な順序で配送し、関係がなければメッセージを順序に関係なく配送する。

メッセージの因果順序性の例を図3.17に示す。この例では、送信プロセス

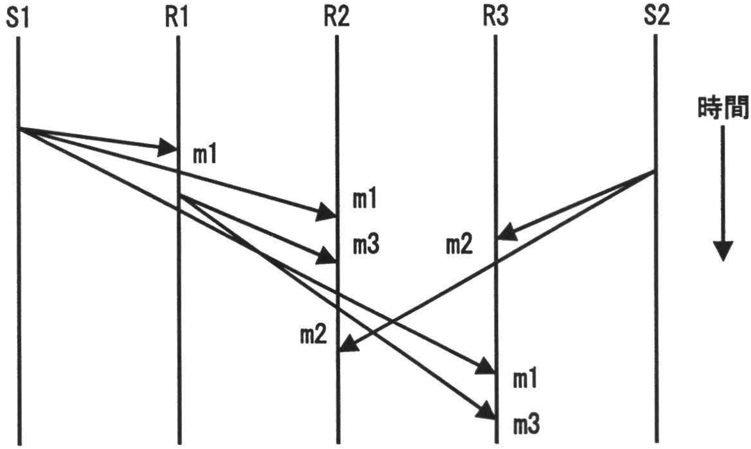


図 3.17 メッセージの因果順序性

S_1 はメッセージ m_1 を受信プロセス R_1、R_2、R_3 に送信し、送信プロセス S_2 はメッセージ m_2 を受信プロセス R_2と R_3 に送信する。m_1 を受信した R_1 は新しいメッセージ m_3 を生成し、受信プロセス R_2と R_3 に送信する。ここで、m_3 送信の事象は、m_3 の内容が m_1 によって作成されているため、m_1 送信の事象と因果関係を持つ。したがって、2 つのメッセージは m_1、m_3 の順序で R_2 と R_3 に配送されなければならない。また、m_2は m_1と m_3 に因果関係を持たないため、m_2は m_1と m_3 とは無関係に R_2と R_3 に配送できる。このことは図 3.17 の例で分かる。

因果順序性の実現の 1 つの方法は、ISIS システムの **CBCAST** プロトコル **(CBCAST protocol)** である [Birman et al. 1991]。これは次のように動作する。

1. グループの各メンバープロセスは、n 要素のベクタを管理する。ここで、n はグループのメンバー数である。各メンバーは順序番号 0 から n が割り当てられ、ベクタの i 番目の要素は順序番号 i のメンバーに対応する。特に、メンバーのベクタ i 番目の要素の値はメンバー i からのこのメンバーによって順番に受信した最後のメッセージの番号に等しい。

2. メッセージを送信するために、プロセスはそれ自身のベクタのその要素の

値を1加算し、メッセージの1部としてベクタを送信する。

3. メッセージが受信プロセスのサイトに届くと、実行時のシステムによってバッファに入れられる。実行時のシステムは、メッセージがユーザプロセスに配送できるかどうかまたはその配送が因果順序性を保証するために遅らせなければならないかを決めるために下記に与えられた2つの条件を調べる。Sをメッセージに付加された送信プロセスのベクタとし、Rを受信プロセスのベクタとする。また、iを送信プロセスの順序番号とする。調べられる2つの条件は次のものである。

$S[i] = R[i] + 1$　かつ　$S[j] \leq R[j]$、ここで全てのjはiと等しくない。

最初の条件は、受信プロセスが送信プロセスからのメッセージを取りこぼしていないことを保証する。この確認は、同じ送信プロセスからの2つのメッセージが必ず因果順序性を持つためには必要である。2番目の条件は、送信プロセスが受信プロセスがまだ受信していないメッセージを受信していないことを保証する。この確認は、送信プロセスのメッセージが受信プロセスによって受信できなかったメッセージと因果関係を持たないことを保証するためには必要である。

メッセージがこれらの条件を満たせば、実行時のシステムはこのメッセージをユーザプロセスを配送する。満足しない場合には、そのメッセージはバッファに置かれ、新しいメッセージが到着した時に再び条件を調べる。

このアルゴリズムを示すための簡単な例を図3.18に示す。この例では、4つのプロセスA、B、C、Dがある。ある時点でのこれらのベクタの状態は、それぞれ(3、2、5、1)、(3、2、5、1)、(2、2、5、1)、(3、2、4、1)となる。これは、これまでに、Aは3つのメッセージを送信し、Bは2つのメッセージを送信し、Cは5つのメッセージを送信し、Dは1つのメッセージを他のプロセスに送信したことを示している。ここで、Aは新しいメッセージを他のプロセスに送信する。したがって、メッセージに付与されるベクタは(4、2、5、1)である。このメッセージは、2つの条件を満足するためにBに配送される。しかしながら、プロセスCのサイトでは最初の条件を、プロセスDのサイトでは2番目の条件を満足しないために、プロセスCとDのサイトの実行時のシステムによってこのメッセージは遅らされなければならない。

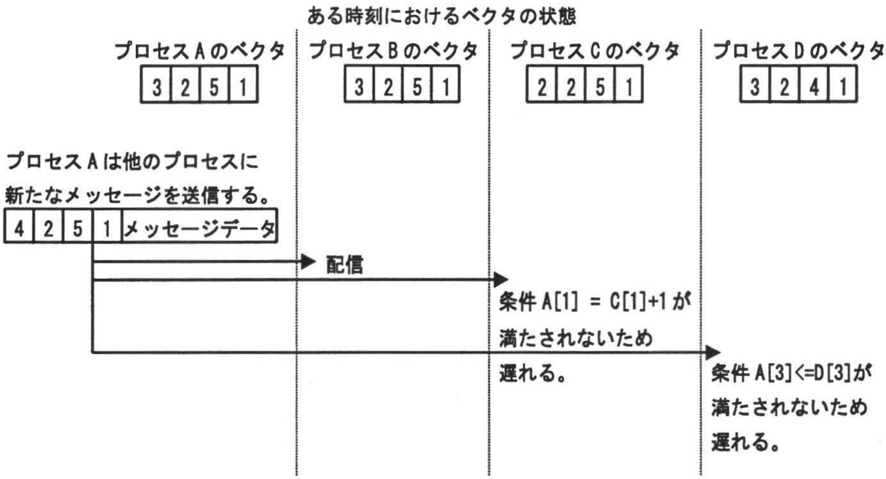


図 **3.18**　因果関係のあるセマンティクスを実現する CBCAST プロトコルの例

良いメッセージパッシングシステムは、少なくとも一貫順序性と因果順序性を持ち、かつユーザがアプリケーションとしてどちらかを選択できる柔軟性を提供すべきである。

3.11　事例：4.3BSD UNIX の IPC 機構

4.3BSD UNIX システムのソケットに基づく IPC は、本章で述べた概念と機構を使ってメッセージパッシングシステムがどのように設計されているかを示している。このシステムは、カリフォルニア大学のバークレー校のコンピュータシステム研究グループ (CSRG) によって開発され、最も幅広く使用されかつ資料化されているメッセージパッシングシステムである。

3.11.1　基本概念と主要機能

4.3BSD UNIX の IPC 機構は、ネットワークに基づくアプリケーションに対して汎用的なインタフェースを提供する。この基本的概念と主要機能を次に述べる。

1. 4.3BSD UNIX の IPC 機構は、異なるプロトコル、名前付け方式及びハー

ドウェアなどを使用する通信ネットワークを実現できることからネットワークに依存しない。このため、4.3BSD UNIX の IPC 機構は通信機能の標準集合として扱うための**通信ドメイン (communication domain)** と呼ぶ用語を使用する。本章では、既に通信エンドポイントを表す各種の名前付け方法を説明した。また、通信に関する各種の同期方式、信頼性、順序性などについても述べた。ネットワークが異なれば、名前付け方式も異なり、通信エンドポイントの表記も変わり、通信の機能を異なってくる。これらのネットワークの特性は、**通信特性 (communication properties)** と呼ばれる。同じ通信特性を持つネットワークは、共通の通信ドメイン (又はプロトコルファミリ) に所属する。通信プリミティブのパラメータとして通信ドメインを指定できる柔軟性を提供することで、4.3BSD UNIX の IPC 機構はユーザにアプリケーションに適した通信ドメインを選択することを可能にしている。

2. 4.3BSD UNIX の IPC 機構は、通信のエンドポイントに対して**ソケット (socket)** と呼ばれる統一的な抽象的概念を使用する。すなわち、ソケットはメッセージを送信又は受信するための抽象的なオブジェクトである。IPC 動作は、ソケットの対に基づいており、それらは同じ若しくは異なるコンピュータの対となる通信プロセスに属する。このソケットの対は、2 つのプロセス間の一方向又は両方向通信に使用される。送信プロセスが送信したメッセージは、ネットワークプロトコルによってネットワークに転送され、そして確認通知を受信する (プロトコルが要求した時にのみ) までソケットに置かれる。受信プロセス側では、メッセージは受信プロセスがそのメッセージを受信するためのシステム呼び出しを行うまで受信プロセスのソケットに置かれる。

プロセスは、他のプロセスと通信行うためにソケットを生成できる。生成されたソケットは、削除されるか、それを参照する全てのプロセスが終了するまで存在する。

3. 位置透過性のために、4.3BSD UNIX の IPC 機構は通信エンドポイントを表現するために 2 レベルの名前方式を使用する。すなわち、ソケットに人間が読める文字列である高位レベル名を割り当てることができる。ソケットの低位レベル名は、通信ドメインに依存する。例えば、ローカルのポート番号とインターネットアドレスから構成される。高位レベルソケット名から低位レベル名に変換するために、4.3BSD はカーネルに置かれる変換機能よりもむしろアプリケーションに提供する。ソケットの高位レベル名は、ソケットが生成された

通信ドメイン内でのみ有効である。

4. 4.3BSD UNIX の IPC 機構は、制御又は統一した方法でアプリケーションに通信のセマンティクスを提供するためにソケットに対してタイプの機構を使用するため高い柔軟性を持っている。すなわち、全てのソケットは通信セマンティクス、例えば順序配送、非重複配送、信頼配送、コネクションレス型通信、コネクション型通信などにしたがってタイプが決められる。システムは、幾つかの標準のソケットタイプを定義し、またユーザが必要に応じてユーザ固有のソケットタイプを定義できる柔軟性を提供する。例えば、ソケットタイプとしてデータグラムは信頼性のないコネクションレス型のパケット通信を、ソケットタイプとしてストリームは信頼性のあるコネクション型のバイトストリームをモデル化している。

5. メッセージは、下位のネットワークがブロードキャスト機能を提供できる場合にはブロードキャストできる。

3.11.2　IPCプリミティブ

4.3BSD UNIX の IPC 機構のプリミティブは、TCP、UDP などのネットワークプロトコルの上位層のシステム呼び出しとして提供される。ネットワークプロトコルの上位に直接 IPC 機構を実現することは、その有効性を高める。最も重要で利用可能な IPC プリミティブの概要を次に説明する。

s = socket(domain, type, protocol)

プロセスが他のプロセスと通信するためには、まず socket システム呼び出しを使用してソケットを生成する。この呼び出しの最初のパラメータ domain は通信ドメインを指定する。最も共通的に使用されるドメインは、多くのホストがインターネット通信ドメインに対応しているためインターネット通信ドメインである。2 番目のパラメータ type はアプリケーションに要求される通信セマンティクスに応じて選択されるパケットタイプを指定する。3 番目のパラメータ protocol はソケットが動作する通信プロトコル (例えば、TCP/IP 又は UDP/IP) を指定する。このパラメータが 0 であればシステムが適切なプロトコルを設定する。socket システム呼び出しは、ソケットに対して引き続くシ

ステム呼び出しを参照するために記述子を返す。生成されたソケットは通常のcloseシステム呼び出しで終了する。

bind(s, addr, addrlen)

ソケットを生成すると、受信プロセスはソケットアドレスを付与する。両方向通信を2つのプロセス間で行う場合、両プロセスがメッセージを受信するためそれぞれがソケットにソケットアドレスを付与しなければならない。このためにbindシステム呼び出しが使用される。この呼び出しの3つのパラメータは生成されたソケットの記述子sと、ソケットアドレスaddrとそのバイト長addrlenを含む。

ソケット生成のシステム呼び出しとソケットアドレス(名前)を付与するシステム呼び出しを1つのシステム呼び出しとして組み合わせることも考えられる。これらのシステム呼び出しを別々にしている理由として大きくは2つある。1つは、ソケットは名前がなくても有効である。生成したソケットに必ず名前を与えることは、意味のない名前を与えることもありえる。もう1つは、ある通信ドメインはソケットの名前を付与する時に付加的な標準化されていない情報(例えばサービスタイプ)を要求するためである。ソケット生成時にこの情報を与えることはインタフェースを複雑にすることになる。

connect(s, server_addr, server_addrlen)

4.3BSD UNIXのIPC機構で最も使用される2つの共通の通信タイプは、コネクション型(ストリーム)通信とコネクションレス型(データグラム)通信である。コネクション型通信では、2つのプロセスは2つのソケット間にコネクションをまず確立する。コネクション確立処理は、一方のプロセスが確立要求を受けるために待ち、他方のプロセスが確立要求を出すため非対称である。コネクションが確立すると、2つのプロセス間に両方向でデータを転送できる。このタイプの通信は、クライアントサーバアプリケーションの実現に有効である。サーバはソケットを生成し、名前を付与し、この名前を広く知らせる。そして、クライアントプロセスからのコネクション要求を待つ。クライアントはコネクション要求をサーバに送信する。コネクションが確立すると、要求と応答を更新するコネクション型通信は信頼性のあるメッセージの交信を実現する。

コネクションレス型通信では、ソケットの対は通信を行う毎に指定される。

このために、送信プロセスはメッセージを送信する毎にローカルのソケット記述子と受信プロセスのソケットアドレスを指定する。コネクションレス型通信は、機能的に信頼性が低い。

connect システム呼び出しは、クライアントのソケットと通信を期待するサーバのソケット間にコネクションを確立するための要求をクライアントが送信するコネクション型通信で使用される。この呼び出しの 3 つのパラメータは、クライアントのソケットの記述子 s、サーバのソケットのソケットアドレス server_addr とそのアドレスのバイト長 server_addrlen である。connect 呼び出しは、自動的にソケットアドレス (名前) とクライアントのソケットを結合する。したがって、その前に結合の処理は必要としない。

listen(s, backlog)

listen システム呼び出しは、クライアントからのコネクションの要求を受けるためにサーバプロセスがコネクション型通信のため使用される。この呼び出しの 2 つのパラメータは、サーバソケットの記述子 s と受理可能な最大コネクション数 backlog である。

snew = accept(s, cliant_addr, cliant_addrlen)

accept システム呼び出しは、サーバプロセスがクライアントからのコネクション確立の要求を受理し、かつクライアントとの通信を行うためにクライアントのソケットを得るためにコネクション型通信で使用される。この呼び出しの 3 つのパラメータは、サーバのソケットの記述子 s、クライアントのソケットアドレス cliant_addr とそのアドレスのバイト長 cliant_addrlen である。この呼び出しは、accept 呼び出しの実行で自動的に生成された新しいソケット記述子 (snew) を返す。この新しいソケットはクライアントのソケットと対であり、サーバは他のクライアントからのコネクション要求を受理するために記述子 s を持つ元のソケットを使用し続けることができる。

データ送受信のためのプリミティブ

データの送受信のための各種のプリミティブがある。4 つの共通的なものは次のものである。

nbytes = read(snew, buffer, amount)
write(s, "message", msg_length)
amount = recvfrom(s, buffer, sender_address)
sendto(s, "message", receiver_address)

read システム呼び出しと write システム呼び出しは、コネクション型通信に最も適したものである。write 動作は、クライアントがサーバにメッセージを送信するために使用する。メッセージを送信するために使用するソケット、メッセージ及びメッセージ長はパラメータとして指定される。read 動作は、サーバプロセスがクライアントの送信したメッセージを受信するために使用する。クライアントのソケットと関係するサーバのソケットと受信するメッセージを受けるバッファは、パラメータとして指定される。この呼び出しは、実際に受信したデータ量を返す。クライアントとサーバ間のソケットのコネクション確立は、メッセージの境界を持たないストリームデータのチャネルのように振る舞う。すなわち、送信プロセスはデータをチャネルに押し出し、受信プロセスは対応する送信動作で書かれた同じシーケンスでデータを読み。チャネルのサイズは、受信ソケットのキューの大きさに制限される。キューが一杯になると送信プロセスは停止し、キューがなければ受信プロセスは停止する。

一方、recvfrom と sendto システム呼び出しは、コネクションレス型通信に最も適している。sendto 動作は、送信プロセスがメッセージを特定の受信プロセスに送信するために使用される。メッセージが送信されるソケット、メッセージ及び受信プロセスのソケットアドレスはパラメータとして指定される。recvfrom 動作は、受信プロセスが特定の送信プロセスからメッセージを受信するために使用される。受信するためのソケット、メッセージを受けるバッファ及び送信プロセスのソケットアドレスは、パラメータとして指定される。recvfrom 呼び出しは、ソケットにキューされている最初のメッセージを受信する。しかしながら、キューが空であればメッセージが到着するまで受信プロセスは停止する。

図 3.19 は、2 つのプロセス間のコネクションレス型通信のソケットの使用を示す。socket 呼び出しでは、最初のパラメータの *AF_INET* は通信ドメインがインターネット通信ドメインであることを、2 番目のパラメータの *SOCK_DGRAM* はデータグラムタイプのソケットを示す (信頼性のないコネクションレス型通信に使用)。

一方、図 3.20 はクライアントプロセスとサーバプロセス間のコネクショ

```
.
.
s = socket(AF_INET, SOCK_DGRAM, 0);
.
.
bind(s, sender_address, server_address_length);
.
.
sendto(s, "message", receiver_address);
.
.
close(s);
```

(a)送信側プログラムのソケットによるシステム呼び出し

```
.
.
s = socket(AF_INET, SOCK_DGRAM, 0);
bind(s, receiver_address, receiver_address_length);
.
.
amount = recvfrom(s, buffer, sender_address);
.
.
close(s);
```

(b) 受信側プログラムのソケットによるシステム呼び出し

図 3.19　2 つのプロセス間のコネクションレス型通信でのソケットの使用

```
.
.
s = socket(AF_INET, SOCK_STREAM, 0);
.
.
connect(s, server_address, server_address_length);
.
.
write(s, "message", msg_length);
.
.
close(s);
```

(a) クライアントプログラムのソケットによるシステム呼び出し

```
.
.
s = socket(AF_INET, SOCK_STREAM, 0);
bind(s, server_address, server_address_length);
listen(s, backlog);
.
.
snew = accept(s, client_address, client_address_length);
.
.
nbytes = read(snew, buffer, amount);
.
.
close(snew);
close(s);
```

(b) サーバプログラムのソケットによるシステム呼び出し

図 3.20　クライアントとサーバ間のコネクション型通信でのソケットの使用

ン型通信のソケットの使用を示す。socket 呼び出しの 2 番目のパラメータの *SOCK_STREAM* は、ストリームタイプのソケットを示す (信頼性のあるコネクション型通信に使用)。

3.12 まとめ

プロセス間通信 (IPC) は、2 つ以上のプロセスでの情報の共有を要求する。この情報の共有に対する 2 つの基本的な方法は、元のデータの共有 (共有データの方法) とコピーデータの共有 (メッセージパッシングの方法) である。ネットワーク上のコンピュータはメモリを共有することができないため、分散システムではメッセージパッシングの方法がもっとも共通的に使用される。

メッセージパッシングシステムは、メッセージに基づくプロトコルを提供する分散オペレーティングシステムのサブシステムであり、複雑なネットワークプロトコルの詳細や複数の不均質なプラットフォームの詳細をプログラマから隠蔽する。

良いメッセージパッシングシステムに要求される機能は、単純性、統一したセマンティクス、有効性、信頼性、正確性、柔軟性、セキュリティ及び移植性である。

メッセージの送信プロセスと受信プロセスは、同期または非同期モードで通信を行う。同期通信に比べて、非同期モードはより高い並行性、低メッセージトラヒック及び高い柔軟性を提供する。しかしながら、非同期通信は実現が複雑であり、メッセージバッファを必要とし、プログラマに対して従来の集中型プログラミング方式にはないものである。

IPC 機構を設計する上で、4 つのタイプのバッファ方式として、ヌルバッファ (又は無バッファリング)、単一メッセージバッファ、無制限容量バッファ及び制限バッファ(又は多重メッセージバッファ) がある。メッセージは、パケットの形式で伝送チャネル上を転送される。したがって、最大パケット長を超えるメッセージを転送するためには、メッセージは分割され、複数のパケットとして送信される。このようなメッセージは、マルチパケット又はマルチデータグラムメッセージと呼ばれる。

符号化は、メッセージのプログラムオブジェクトを伝送チャネルに転送するために適したストリーム形式に変換する処理である。この処理はメッセージを送信する側で行われる。受信側でメッセージデータからプログラムオブジェクトを再構成する逆の処理は、メッセージデータの復号化と呼ばれる。

メッセージパッシングの別の重要な課題は、相互作用を行う対のアドレス付け(名前付け)である。プロセスは、特定のプロセス又は特定のタイプの任意のプロセスと通信を行いたいか否かによって通信の相手のプロセスを明示的に指定することもあればそうしない場合もある。プロセスアドレス付けの重要な目標は、位置透過性を提供することである。このためには、プロセスに対して2レベル名前付け方式が使用され、上位レベルの装置非依存のプロセス名から低位レベルの装置依存の名前に対応づける名前サーバが使用される。

障害処理は、IPC機構を設計する上での重要な課題である。信頼性のあるIPCプロトコルを設計するための2つの共通的な方法は、タイムアウトによる再送と明示的な確認通知パケットの使用である。IPC機構における障害処理の2つの重要な課題は、繰り返し等価と、マルチデータグラムメッセージにおける要求メッセージの重複処理とパケットの損失と順序誤りの復旧処理である。

メッセージに基づく相互作用の最も基本的な形態は、1つの送信プロセスが1つの受信プロセスにメッセージを送信する1対1通信である。しかしながら、性能や柔軟性を高めるために、幾つかの分散システムでは送信プロセスと受信プロセス間において1対多、多対1及び多対多の相互作用を可能とするグループ通信機能を提供する。グループ通信に関する課題は、グループ管理、グループアドレス付け、原子性、及び順序メッセージ配送である。

3.13　演習問題

3.1 メッセージパッシングシステムを設計する上で主な信頼性の課題は何か？課題に対する適切な機構を述べよ。

3.2 メッセージパッシングシステムのIPCプロトコルの正確性に関する主な課題は何か？ 課題を処理するための適切な機構を述べよ。

3.3 メッセージパッシングシステムがユーザに提供すべき柔軟性の機能を述べよ。柔軟性の機能の利点を利用するための適切なIPCプリミティブを示せ。

3.4 IPCのブロッキングと非ブロッキングタイプを述べよ。どちらが実現しやすいか、また何故か？ これに関連する利点と欠点を述べよ。

3.5 次の2つの場合に対する生産者プロセスと消費者プロセスの対の実現方法を記述せよ。

(a) 単一バッファを使用する。

(b)n メッセージ分に相当するバッファが使用できる。

生産者はメッセージを生成し、メッセージバッファに入れる。消費者はメッ

セージバッファからメッセージを消費する。全てのバッファは固定長とする。

3.6 データグラムとは何か？ 何故マルチデータグラムメッセージは IPC で使用されるか？ マルチデータグラムメッセージの IPC での主な課題は何か？ 課題を処理するための適切な機構を述べよ。

3.7 マルチデータグラム通信で、受信者は同じメッセージのデータグラムをどのように認識し、処理するか、またメッセージの最後のデータグラムはどのように認識できるかを説明せよ。

3.8 文字がリンクされたリストはバイトストリームの形式でネットワークを転送される。送信側でのデータ構造の符号化と受信側での復号化のためのプログラミング言語による実現方法を記述せよ。

3.9 1 対多通信における掲示板セマンティクスを実現する方法において次の 2 つの場合について記述せよ。

(a) 掲示板に書かれたメッセージは n 受信者が読むと自動的に削除される。

(b) 掲示板に書かれたメッセージは時間 t が経過すると自動的に削除される。時間 t は送信者によって指定される。

同時には 1 つのメッセージのみが掲示板に書かれる。メッセージを送信したい時に掲示板が空きでない場合、送信者に対して「空きなし。再度送れ」のメッセージを送信者に返す。必要な前提条件があれば明確に定義して説明せよ。

3.10 下記に示すマルチキャスト通信のタイプに最適なアプリケーションの例をそれぞれ 2 つ示せ。

(a)0 信頼性

(b)1 信頼性

(c)n 中 m 信頼性 ($1 < m < n$)

(d) 全信頼性

3.11 「順序メッセージ配送」とは何か？ 全てのアプリケーションは同じセマンティクスを必要とするか？ もしそうであれば、理由を述べよ。そうでなければ、異なるセマンティクスを必要とするアプリケーションの例を 2 つ示せ。

3.12 絶対順序性、一貫順序性及び因果順序性とは何か説明せよ。それぞれの実現するための機構を示せ。

3.13 下記の場合におけるメッセージの一貫順序性の実現のための機構を述べよ。

(a)1 対多通信

(b) 多対 1 通信

(c) 多対多通信

3.14 3 つの異なるプロセスアドレス付け機構を述べよ。これらの相対的な利点と欠点を述べよ。下記の場合においてどのプロセスアドレス付けが最適化を示せ (理由も付けて)。

(a) サーバプロセスと幾つかのクライアントプロセス間の通信で、クライアントプロセスがサーバプロセスに要求メッセージを送り、サーバプロセスは各クライアントプロセスに応答を返す。

(b) 送信プロセスはプロセスのグループにメッセージを送信する。

(c) 送信プロセスは他のノードに移動する受信プロセスにメッセージを送信する。

(d) 送信プロセスは他のノードに移動し、送信プロセスに応答を返す受信プロセスにメッセージを送信する。

(e) クライアントプロセスは幾つかのサーバプロセス中の 1 つからサービスを受ける。

3.15 繰り返し等価な動作はとは何か？ 次の動作のどれが繰り返し等価か？

(a)read_next_record(filename)

(b)read_record(filename, record_number)

(c)append_record(filename, record)

(d)write_record(filename, after_record_n, record)

(e)seek(filename, position)

(f)add(integer_1, integer_2)

(g)increment(variable_name)

3.16 IPC セマンティクスの下記のタイプを実現するための適切な機構を示せ。

(a) 最後の 1 回

(b) 最低 1 回

(c) ちょうど 1 回

3.17 サーバによって実行される動作は、非繰り返し等価である。この場合のちょうど 1 回 IPC セマンティクスの機構を述べよ。

3.18 最低 1 回又はちょうど 1 回セマンティクスのどちらが次のアプリケーションに使用されるかを示せ (理由を付けて)。

(a) ファイルサーバがファイルを読むための要求

(b) ファイルサーバが既存のファイルにあるデータを追加するための要求

(c) 編集サーバがファイルをコンパイルするための要求

(d) データベースサーバが銀行の預金を更新するための要求

(e) データベースサーバが銀行の預金の現在の残高を入手するための要求

(f) 予約サーバが既に予約している座席をキャンセルするための要求

3.19 次の場合について、ちょうど1回セマンティクスの信頼性のあるIPCを実現するための適切な機構を示せ。

(a) 送信と受信プロセスのコンピュータは信頼性があるが、接続する通信リンクは信頼性がない。

(b) 送信と受信プロセスのコンピュータは信頼性がないが、接続する通信リンクは信頼性がある。

(c) 送信プロセスのコンピュータと接続する通信リンクは信頼性があるが、受信プロセスのコンピュータは信頼性がない。

(d) 受信プロセスのコンピュータと接続する通信リンクは信頼性があるが、送信プロセスのコンピュータは信頼性がない。

3.20 メッセージの送信プロセスはメッセージが宛先に正しく届いたことを必ず知る必要があるか？ 必要であれば、その理由を説明せよ。必要でなければ、2つの事例を示せ。

3.21 IPCプロトコルで確認通知メッセージを使用する主な目的は何か？ 信頼性のある通信のために確認通知メッセージは必ず必要か？ 理由も述べよ。

3.22 メッセージの「重畳」とは何か？ ネットワークトラヒックを軽減するためにどのように作用するか？ 分散システムで性能の向上を行うために重畳方式が使用される幾つかの例を示せ。

3.23 多くのクライアントサーバシステムでは、サーバの障害のためにクライアントが応答を待ちつづけることを避けるためにタイムアウト機構が使用される。すなわち、このようなシステムでサーバに要求メッセージを送信後、クライアントがある時間内にサーバから応答を受信しなければ、クライアントはサーバが障害になったと想定し、必要な対応を取ることができる。このようなシステムにおいて、タイムアウトの理想的な時間はどのようにすべきか？ サーバが十分に信頼性がある場合にもタイムアウト機構を使用することは有効か？ 理由も述べよ。

3.24 ファイルサーバは複数のクライアントのリード／ライトの要求を処理する。クライアントはメッセージを送信し、応答を受けることで直接サーバと通信することができる。

(a) クライアントがファイルの一部分を読むためのメッセージの内容を述べよ。

(b) 上記(a)の応答をサーバがクライアントに返すメッセージの内容を述べよ。

(c) クライアントがファイルサーバの既存ファイルにデータを書き込むため

のメッセージの内容を述べよ。

(d) 上記 (c) の応答をサーバがクライアントに返すメッセージの内容を述べよ。

(e) ファイルサーバが複数のクライアントからほとんど同時に要求を処理するための機構を述べよ。

3.25 クライアントサーバ IPC では、クライアントはサーバに要求メッセージを送信し、サーバはその要求を処理してクライアントに要求の処理結果を返す。1 つのプロセスがクライアントとサーバを兼ねることは有効か？ そうでなければ、その理由を述べよ。有効であれば、例を示せ。

3.26 ファイル格納検索システムは、複数のクライアントプロセス、ファイル管理プロセス、キャッシュ管理プロセス及び複数のディスク管理プロセスから構成されるものとする。この時これらのプロセスは次の動作を行う。

(a) クライアントは、ファイル管理プロセスにファイルアクセス要求を送る。

(b) ファイル管理プロセスは、クライアントからの要求を受けるとキャッシュ管理プロセスに要求を送る。

(c) データがキャッシュにあれば、キャッシュ管理プロセスはキャッシュから取り出し、ファイル管理プロセスに応答として返す。なければ、キャッシュ管理プロセスは全てのディスク管理プロセスにその要求を転送する。

(d) キャッシュ管理プロセスから要求を受けると、ディスク管理プロセスはディスク内にデータがあるかを検索し、キャッシュ管理プロセスに応答を返す。すなわち、データがなければ「データ無」のメッセージが返され、見つかれば要求されたデータが返される。

(e) データを受けると、キャッシュ管理プロセスはキャッシュにデータを格納し、ファイル管理プロセスにデータを送る。

(f) ファイル管理プロセスは、データに基づいて要求された処理を行い、最終的に応答をクライアントに送信する。

このアプリケーションでの各プロセスタイプの相互作用でどのような IPC の形式が使用されるか？ これらのプロセス間のインタフェースを定義するための適切な IPC プリミティブを示せ。

3.27 上記の 3.26 の問題のファイル格納検索システムでプロセスがどのように通信するかを示すために 4.3BSD UNIX の IPC 機構のプリミティブを使用してプロセスの基本構成を示せ。

参考文献

[Aveysundara and Kamal 1991] Abeysundara, B. W., and Kamal, A. E., "High-Speed Local Area Networks and Their Performance: A Suravey," *ACM Computing Suraveys*, Vol. 23, No. 2, pp.221-264 (1991).

[ATM Forum 1993] *ATM Use-Network Interface Specification Version 3.0*, Prentice-Hall, Englewood Cliffs, NJ (1993).

[ATM Forum 1994] "LAN Emulation Over ATM," Draft Specification—Revision 5 (ATM FORUM 94-0035R5), LAN Emulation Sub-Working Group of the ATM Forum Technical Committee (August 1994).

[Black 1993] Black, U., *Computer Networks: Protocols, Standards, and Interface*, 2nd ed., Prentice-Hall, Englewood Cliffs, NJ (1993).

[Black 1995a] Black, U., *ATM: Roundation for Broadband Networks*, Prentice-Hall, Englewood Cliffs, NJ (1995).

[Black 1995b] Black, U., *TCP/IP and Related Protocols*, 2nd ed., IEEE Computer Society Press, Los Alamitos, CA (1995).

[Brazdziunas 1994] Brazdziunas, C., "Iping Support for ATM Services," Internet RFC No. 1680 (1994).

[Chao et al. 1994] Chao, H. J., Ghosal, D., Saha, D., and Tripathi, S. K., "IP on ATM Local Area Networks," *IEEE Communications Magazine*, pp.52-59, New York, NY (August 1994).

[Cheriton 1986] Cheriton, D. R., "VMTP: A Transport Protocol for the Next Generation of Communication Systems," In: *Proceedings of the SIGCOMM'86*, pp.406-415 (August 1986).

[Cheriton 1988] Cheriton, D. R., "The V Distributed System," *Communications of the ACM*, Vol. 31, No. 3, pp.314-333, ACM, New York, NY (1988).

[Cheriton and Williamson 1989] Cheriton, D. R., and Williamson, C.L., "VMTP As the Transport Layer for High-Performance Distributed Systems," *IEEE Communications Magazine*, Vol. 27, No. 6, pp.37-44, New York, NY (1989).

[Comer 1995] Comer, D. E., *Internetworking with TCP/IP: Volume1 — Principles, Protocols, and Architectures*, 3rd ed., Prentice-Hall, Englewood Cliffs, NJ (1995).

[**Comer and Stevens 1993**] Comer, D. E., and Stevens, D. L., *Internetworking with TCP/IP: Volume3 — Client-Server Programming and Applications: BSD Socket Version*, Prentice-Hall, Englewood Cliffs, NJ (1993).

[**Comer and Stevens 1994**] Comer, D. E., and Stevens, D. L.,*Internetworking with TCP/IP: Volume 2 — Design. Implementation. And Internals*, 2nd ed., Prentice-Hall, Englewood Cliffs, NJ (1994).

[**Datapro 1990**] Datapro, "An Overview of Simple Network Management Protocol," Datapro Network Management. NM40-300-201 (February 1990).

[**Datapro 1993**] Open Software Foundation (OSF) Distributed Management Environment (DME), A Datapro Report, Datapro Network Management, NM40-684-07(April 1993).

[**DePrycker 1993**] DePrycker, M., *Asynchronous Transfer Mode: Solution for Broadband ISDN*, 2nd ed., Ellis Horwood (1993).

[**DePrycker et al.1993**] DePrycker, M., Peschi, R., and Landegem, T., "B-ISDN and the OSI Protocol Reference Model," *IEEE Network*, Vol. 7, No. 2, pp.10-18, New York, NY (March 1993).

[**Dutton and Lenhard 1995**] Dutton, Jr., H., and Lengard, P., *High-Speed Networking Technology: An Introductory Survey*, Prentice-Hall, Englewood Cliffs, NJ(1995).

[**Eckberg 1992**] Eckberg, A., "B-ISDN/ATM Traffic and Congestion Control," *IEEE Network*, Vol. 6, No. 5, pp.28-37, New York, NY (1992).

[**Finlayson et al. 1984**] Finlayson, R., Mann, T., Mogul, J., and Theimer, M., "A Reverase Address Resolution Protocol," RFC No. 903 (June 1984).

[**Fischer et al. 1994**] Fischer, W., Wallmeier, E., Worster, T., Davis, S. P., and Hayter, A., "Data Communications Using ATM: Architectures, Protocols, and Resource Management," *IEEE Communicaions Magazine*, pp.24-33, New York, NY (August 1994).

[**Furht and Milenkovic 1995**] Furht, B., and Milenkovic, M., *Guided Tour of Multimedia Systems and Applications*, IEEE Computer Society Press, Los Alamitos, CA (1995).

[**Haendel et al. 1994**] Haendel, R., Huber, M. N., and Schroeder, S., *ATM Networks Concepts, Protocols, Applications*, Addison-Wesley, Reading, MA(1994).

[Helgert 1991] Helgert, H. J., *Integrated Services Digital Networks: Architectures, Protocols, Standards*, Addison-Wesley, Reading, MA (1991).

[Hon and Suda 1991] Hong, D., and Suda, T., "Congestion Control and Prevention in ATM Networks," *IEEE Network*, Vol. 5, No. 4, pp.10-16, New York, NY (1991).

[Hughes 1994] Hughes, K., "Entering the World-Wide Web: A Guide to Cyberspace,"
ftp:/taurus.cs.nps.navy.mil:/pub/mbmy/world-wide-web-buide.ps.z(1994).

[IEEE 1985a] *Carrier Sense Multiple Access with Collision Detect (CSMA /CD) Access Method and Physical Layer Specifications*, ANSI/IEEE 802.3 (ISO/DIS 8802/3), IEEE, New York (1985).

[IEEE 1985b] *Token-Passing Bus Access Method and Physical Layer Specifications*, ANSI/IEEE 802.4 (ISO/DIS 8802/4), IEEE, New York (1985).

[IEEE 1985c] *Token Ring Access Method and Physical Layer Specifications*, ANSI/IEEE 802.5 (ISO/DIS 8802/5), IEEE, New York (1985).

[IEEE 1990] *IEEE Standard 802: Overview and Architecture, American National Standard ANSI/IEEE 802*, IEEE Computer Society Press, Los Alamitos, CA (1990).

[IT 1990] Information Technology—Open Systems Interconnection—Management Information Protocol Specification—Common Management Information Protocol, ISO/IEC 9596-1, ISO/IEC JTC1/SC21 N5303 (November 1990).

[Janet 1993] Janet, E. L., "Selecting a Network Management Protocol, Functional Superiority vs. Popular Appeal," *Telephony* (November 1993).

[Kaashoek et al. 1993] Kaashoek, M. F., Van Renesse, R., Staveren, H., and Tanenbaum, A. S., *Computer Systems*, Vol. 11, No. 1, pp.73-106 (1993). ©ACM, Inc., 1993.

[Kawarasaki and Jabbari 1991] Kawarasaki, M., and Jabbari, B., "B-ISDN Architecture and Protocol," *IEEE Journal of Selected Areas on Communications*, Vol. SAC-9, No. 9, pp.1405-1415, New York, NY (1991).

[Kim and Wang 1995] Kim, B. G., and Wang, P., "ATM Network: Goals and Challenges," *Communications of the ACM*, Vol. 38, No. 2, pp.39-44 (1995).

[King and Mitrani 1987] King, P. J. B., and Mitrani, I., "Modeling a Slotted Ring Local Area Network," *IEEE Transactions on Computers*, Vol. C-36, No. 5, pp.554-561, Piscataway, NJ (1987).

[Kleinrock 1992] Kleinrock, L., "The Latency/Bandwidth Tradeoff in Gigabit Networks," *IEEE Communications Magazine*, Vol. 30, No. 4, pp.36-40, New York, NY(1992).

[Krol 1994] Krol, E., *The Whole Internet: User's Guide and Catalog*, 2nd ed., O'Reilly, Sebastopol, CA (1994).

[Kung 1992] Kung, H. T., "Gigabit Local Area Networks: A Systems Perspective," *IEEE Communications Magazine*, Vol. 30, No. 4, pp.79-89, New York, NY (1992).

[Larmouth 1993] Larmouth, J., *Understanding OSI*, Prentice-Hall, London, UK (1993).

[Laubach 1994] Laubach, M., "Classical IP and ARP over ATM," RFC No. 1577 (January 1994).

[Leinwand and Conroy 1996] Leinwand, A., and Conroy, K. f., *Network Management: A Practical Perspective*, 2nd ed., Addison-Wesley, Reading, MA (1996).

[Macedonia and Brutzman 1994] Macedonia, M. R., and Brutzman, D. P., "Mbone Provides Audio and Video Across the Internet," *IEEE Computer*, pp.30-36 (April 1994).

[Malamud 1992] Malamud, C., STACKS: *Interoperability in Today's Computer Networks*, Prentice-Hall, Englewood Cliffs, NJ (1992).

[Malamud 1993] Malamud, C., *Exploring the Internet: A Technical Travelogue*, Prentice-Hall, Englewood Cliffs, NJ (1993).

[Martin 1993] Martin, J. L., "Travels with Gopher," *IEEE Computer*, Vol 26., No. 5,pp.84-87 (1993).

[Miller 1995] Miller, M. A., *Inter Networking: A Guide to Network Communications LAN to LAN; LAN to WAN*, 2nd ed., M&T Books, New York, NY (1995).

[Mosaic 1994] "What's New with NCSA Mosaic," http:www.ncsa.uiuc.edu/SDG/Software/Mosaic/Docs/Watsnew.html (June 1994).

[**Mullender et al. 1990**] Mullender, S. J., Van Rossum, G., Tanenbaum, A. S., Van Renesse, R., and Van Staverene, H., "Amoeba: A Distributed Operating System for the 1990s," *IEEE Computer*, Vol.23., No. 5, pp.44-53(1990).

[**Nejmeh 1994**] Nejmeh, B. A., "Internet: A Strategic Tool for the Software Enterprise," *Communications of the ACM*, Vol. 37, No. 11, pp.23-27(1994). ©ACM, Inc., 1994.

[**Newman 1994**] Newman, P., "ATM Local Area Networks," *IEEE Communications Magazine*, Vol. 32, No.3, pp.86-98 (1994).

[**Omidyar and Aldridge 1993**] Omidyar, C., and Aldridge, A., "Introduction to SDH/SONET," *IEEE Communications Magazine*, Vol. 31, No. 9, pp.30-33 (1993).

[**Partridge 1994**] Prtridge, C., *Gigabit Networking*, Addison-Wesley, Reading, MA (1994).

[**Perlman 1992**] Perlman, R., *Interconnections: Bridges and Routers*, Addison-Wesley, Reading MA (1992).

[**Pllummer 1982**] Plummer, D. C., "An Ethernet Address Resolution Protocol," RFC No. 826 (November 1982).

[**Postel 1980**] Postel, J., "User Datagram Protocol," RFC No. 768, USC Information Sciences Institute (August 1980).

[**Postel 1981a**] Postel, J., "Internet Protocol: DARPA Internet Program Protocol Specification," RFC No. 791 (September 1981).

[**Postel 1981b**] Postel, J., "Internet Control Message Protocol," RFC No. 792 (September 1981).

[**postel 1981c**] Postel, J., "Transmission Control Protocol: DARPA Internet Program Progocol Specification." RFC No. 793 (September 1981).

[**Press 1994**] Press, L., "Commercialization of the Internet," *Communications of the ACM*, Vol. 37, No. 11, pp.17-21 (1994).

[**Ramos et al. 1996**] Ramos, E., Schroeder, A., and Beheler, A., *Computer Networking Concepts*, Prentice-Hall, Englewood Cliffs, NJ (1996).

[**Rooholamini 1995**] Rooholamini, R., "ATM-Based Multimedia Servers," *IEEE Multimedia*, pp.39-52 (Spring 1995).

[**Santifaller 1994**] Santifaller, M., *TCP/IP and ONC/NFS, Internetworking in a UNIX Environment*, 2nd ed., Addison-Wesley, Reading, MA (1994).

[**Shevenell 1994**] Shevenell, M., "NMPv2 Needs Reworking to Emerge as a Viable Net Management Platform, "*Network World* (March 7, 1994).

[**Smythe 1995**] Smythe, C., *Internetworking: Designing the Right Architectures*, Addison-Wesley, Reading, MA (1995).

[**Stallings 1992a**] Stallings, W. (Ed.), *Advances in ISDN and Broadband ISDN*, IEEE Computer Society Press, Los Alamitos, CA (1992).

[**Stallings 1992b**] Stallings, W., *Computer Communications: Architectures, Protocols and Standards*, 3rd ed., IEEE Computer Society Press, Los Alamitos, CA (1992).

[**Stallings 1993a**] Stallings, w. (Ed.), *Advances in Local and Metropolitan Area Networks*, IEEE Computer Society Press, Los Alamitos, CA (1993).

[**Stallings 1993b**] Stallings, W. (Ed.), *Network Management*, IEEE Computer Society Press, Los Alamitos, CA (1993).

[**Stallings 1993c**] Stallings, W., *Networking Standards: A Guide to OSI, ISDN, LAN, and MAN Standards*, Addison-Wesley, Reading, MA (1993).

[**Stallings 1995**] Stallings, W., *ISDN and Broadband ISDN with Frame Relay and ATM*, 3rd ed., Prentice-Hall, Englewood Cliffs, NJ (1995).

[**Suzuki 1994**] Suzuki, T., "ATM Adaptation Layer Protocol," *IEEE Communications Magazine*, Vol. 32, No. 4, pp.80-83 (1994).

[**Tanenbaum 1988**] Tanenbaum, A. S., *Computer Networks*, 2nd ed., Prentice-Hall, Englewood Cliffs, NJ (1988).

[**Tanenbaum 1995**] Tanenbaum, A. S., *Distributed Operating Systems*, Prentice-Hall, Englewood Cliffs, NJ (1995).

[**Tittel and James 1996**] Tittel, E., and James, S., *ISDN Networking Essentials*, Academic Press, San Diego, CA (1996).

[**Trajkovic and Golestani 1992**] Trafkovic, L., and Golestani, S. J., "Congestion Control for Multimedia Services," *IEEE Network*, Vol. 6, No. 5, pp.20-26 (1992).

[**Turner 1992**] Turner, J., "Managing Bandwidth in ATM Networks with Burstyy Traffic," *IEEE Network*, Vol. 6, No. 5, pp.50-58 (1992).

[Umar 1993] Umar, A., *Distributed Computing: A Practical Approach*, Prentice-Hall, Englewood Cliffs, NJ (1993).

[Verma 1990] Verma, P. K., *ISDN Systems: Architecture, Technology & Applications*, Prentice-Hall, Englewood Cliffs, NJ (1990).

[Vetter 1995] Vetter, R. J., "ATM Concepts: Architectures, and Protocols," *Communications of the ACM*, Vol.38, No. 2, pp.31-38 (1995).

[Vetter and Du 1993] Vetter, R. J., and Du, D. H. C., "Distributed Computing with High-Speed Optical Networks," *IEEE Computer*, Vol. 26, No. 2, pp.8-18 (1993).

[Vetter et al. 1994] Vetter, R. J., Spell, C., and Ward, C., "Mosaic and the World-wide Web," *IEEE*

Computer, pp.49-56 (October 1994).

[Vickers and Suda 1994] Vickers, B. J., and Suda, T., "Connectionless Service for Public ATM Networks," *IEEE Communications Magazine*, pp.34-42 (August 1994).

[Voruganti 1994] Boruganti, R. R., "A global Network Management Framework for the '90s," *IEEE Communications Magazine*, pp.74-83 (August 1994).

[Wilkes and Wheeler 1979] Wilkes, M. V., and Wheeler, D. J., "The Cambridge Digital Communication Ring," In: *Proceedings of the Local Area Communications Network Symposium*, Boston, pp.47-61 (May 1979).

[Wittie 1991] Wittie, L. D., "Computer Networks and Distributed Systems," *IEEE Computer*, Vol. 24, No. 9, pp.67-75 (1991).

[WWW 1994a] "World-Wide Web Growth," ftp:nic.merit.edu (1994).

[WWW 1994b] "The WWW Virtual Library," http:info.cern.ch/hypertext/DataSources/bySubject/Overview.html (June 1994).

[Yazid and Mouftah 1992] Yazid, S., and mouftah, H. T., "Congestion-Control Methods for B-ISDN," *IEEE Communications Magazine*, Vol. 30,No. 7, pp.42-47 (1992).

インターネット参考文献

メッセージパッシングについての参考文献はないが、下記の文献には本章に関するものがある。

ftp:ftp.cs.umanitoba.ca/pub/bibliographies/Os/IMMD_IV.html
ftp:ftp.cs.umanitoba.ca/pub/bibliographies/Os/os.html
ftp:ftp.cs.umanitoba.ca/pub/bibliographies/Parallel/JPDC.html
ftp:ftp.cs.umanitoba.ca/pub/bibliographies/Parallel/pvm.html

第4章

遠隔手続き呼び出し

4.1　概要

プロセス間通信 (IPC) の汎用的なメッセージパッシングモデルは、前章で説明した。分散アプリケーションの IPC 部分について、メッセージパッシングモデルに基づく IPC プロトコルを使用することによって適切にかつ効果的に処理される。しかしながら、個別に開発された IPC プロトコルは、特定のアプリケーション用であるため、各種の分散アプリケーションの共通的な機能を提供しない。したがって、分散アプリケーション共通に設計するために使用できる汎用的な IPC プロトコルの必要性が残されていた。このような要求に対応するものが遠隔手続き呼び出し (RPC) 機能である。これは、IPC の汎用的なメッセージパッシングの特別なケースである。分散システムを構築するためにプログラマに慣れ親しんだ機構を提供することは、RPC 機能を開発する大きな動機の 1 つである。RPC 機能は全ての分散アプリケーションに対する汎用的なものではないが、多くの分散アプリケーションを構築するために適した価値ある通信機構を提供するものである。

RPC は、分散システムにおいて幅広く受け入れられている IPC 機構となっている。分散アプリケーションの主な通信機構として RPC が普及しているのは次の理由による。

1. 単純な呼び出し形式
2. 親しみ易いセマンティクス
3. 理解しやすいインタフェース仕様。この特性によりコンパイル時の型検査

や自動インタフェース生成に使用される。

4. 使用の容易性。明確で単純な手続き呼び出しのセマンティクスにより分散処理が容易に構築でき、かつ正確である。

5. 汎用性。この機能は、単一マシンでの処理手続き呼び出し方式がアルゴリズムを処理する部分の間の通信においても重要な機構となることでも分かる [Birrell and Nelson 1984]。

6. 効果性。手続き呼び出しは通信を高速に処理するに十分単純である。

7. IPC 機構は、同じマシン上のプロセス間通信と同じように異なるマシンのプロセス間通信にも使用できる。

4.2　RPC モデル

RPC モデルは、次のような制御やデータの引く渡し方法で使用される良く知られた手続き呼び出しに良く似ている。

1. 手続き呼び出しでは、呼び出し側は決まった位置にパラメータを設定する。

2. 制御は、手続きの本体の先頭に渡される。

3. 手続きは、呼び出し側から与えられたパラメータをコピーし、新たな実行環境で実行される。

4. 手続きの実行が終わると、結果をもって呼び出した点に制御が戻される。

RPC 機構は、呼び出しプロセスとは異なるアドレス空間にある手続きを呼び出すことができるという点で手続き呼び出し機構の拡張である。呼び出し手続き (**遠隔手続き呼び出し (remote procedure call)** と呼ばれる) は、呼び出しプロセスと同じマシン又は異なるマシンにある。

RPC の場合、呼び出し側と呼ばれる側のプロセスは異なるアドレス空間 (恐らく異なるマシン) にあるため、遠隔手続き呼び出し側の環境にあるデータや変数にアクセスできない。したがって、RPC 機能はこれらのプロセス間の情報の交信のためにメッセージパッシング方式を使用する。図 4.1 に示すように、遠隔手続き呼び出しが行われると、両者のプロセスは次のような相互作用を持つ。

1. 呼び出し側 (**クライアントプロセス (client process)** と呼ばれる) は、呼び出し (要求) メッセージを呼び出し側 (**サーバプロセス (server process)** と呼ばれる) に送信し、応答メッセージを待つ (ブロックされる)。要求メッセー

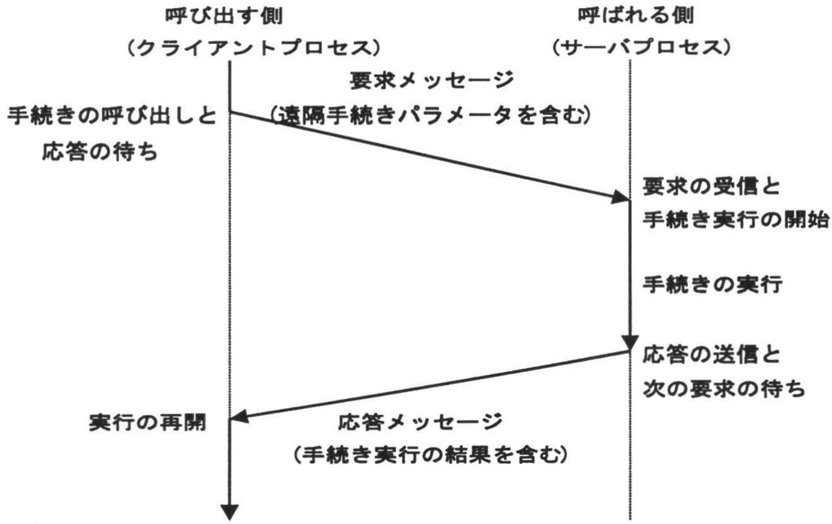


図 **4.1** 遠隔手続き呼び出しの典型的なモデル

ジは遠隔手続きのパラメータを含む。

2. サーバプロセスは手続きを実行し、実行結果を含むメッセージを呼び出し側に返す。

3. 応答メッセージを受けると、実行結果が確認され、呼び出し側の実行が再開する。

サーバプロセスは通常非活性化状態にあり、要求メッセージの到着を待つ。メッセージが到着すると、サーバプロセスは手続きのパラメータを確認して実行を行い、応答を返し、次の呼び出しメッセージを待つ。

RPC モデルにおいては、2 つのプロセスの 1 つのみがある時点では活性化している。しかしながら、一般に、RPC プロトコルは並行性モデルを実現することに対する制限はなく、RPC の他のモデルとして呼び出しが非同期に行われる方式ではクライアントはサーバからの応答が返る間も処理ができる。他の方法として要求を処理するためにサーバがスレッド (第 8 章で説明する) を生成し、他の要求の受信に対応できる。

4.3　RPC の透過性

RPC 機能を設計する上での大きな課題は、**透過性 (transparency)** である。透過的な RPC 機構はローカルの手続きと遠隔手続きがプログラマに対して区別されないことである。このことは、次の 2 つのタイプの透過性を要求する [Wilbur and Bacarisse 1987]。

1. **シンタックス透過性 (syntactic transparency)** は、遠隔手続き呼び出しがローカル手続き呼び出しと同じシンタックスを持つことを意味する。
2. **セマンティクス透過性 (semantic transparency)** は、遠隔手続き呼び出しとローカル手続き呼び出しのセマンティクスが同じであることを意味する。

RPC 機構のシンタックス透過性を達成することはそんなに大変ではなく、またセマンティクスについても遠隔手続き呼び出しをローカル手続き呼び出しのものと類似したものにしている。

■呼び出しプロセスは、呼び出されたプロセスから返されるまで中断する。
■呼び出しプロセスは、呼び出される手続き (遠隔手続き) にパラメータを渡す。
■呼ばれる手続き (遠隔手続き) は、呼び出し側に結果を返すことができる。

しかしながら、遠隔手続き呼び出しとローカル手続き呼び出しのセマンティクスを同じにすることは不可能に近い [Tanenbaum and Van Renesse 1988]。これは、下記に示す遠隔手続き呼び出しとローカル手続き呼び出しの相違による。

1. ローカル手続き呼び出しと異なり、遠隔手続き呼び出しでは呼び出す側のアドレス空間とは異なるアドレス空間で呼び出される手続きが実行される。このため、呼ばれる (遠隔) 手続きは呼び出す側のプログラムの環境のデータや変数にアクセスできない。このように、共有メモリがない場合、パラメータとしてアドレスを渡すことは意味を持たなく、参照パラメータのポインタは要をなさない。同様に、ポインタはメモリアドレスを通常表すため、ポインタ構造 (例えば、リンクされたリスト) を含むパラメータ値を渡すことは意味を持たない。Bal 等 [1989] によると、呼び出し側によって渡されるポインタの整合性は呼び出し側で行われなければならないが、付加的な通信が必要となる。別の方

法としは、受信側にポインタが図形などの複雑なデータ構造をポイントしている場合には実現が困難である。同じように、参照パラメータは、コピーイン／コピーアウトによって置き換えられるが、セマンティクスに相違が生じる。

2. 遠隔手続き呼び出しは、ローカル手続き呼び出しに比べてネットワークや異なるマシンを含むため障害を受け易い。したがって、遠隔手続き呼び出しを行うプログラムはローカル手続き呼び出しでは発生しないこれらの障害に対して対応できる能力が必要になる。マシン障害やネットワークの通信障害への対応は、遠隔手続き呼び出しとローカル手続き呼び出しのセマンティクスを同じにすることをますます困難にする。

3. 遠隔手続き呼び出しは、ローカル手続き呼び出しに比べて多くの時間 (100 から 1000 倍) を必要とする。これは、RPC が通信ネットワーク上で行われるためである。したがって、RPC を使用するアプリケーションはネットワークの輻輳により発生する長い遅延に対処できる能力を持たなければならない。

遠隔手続き呼び出しに通常の呼び出しのセマンティクスを持たせることが困難なため、幾人かの研究者は RPC 機能は非透過性であるべきと考えている。例えば、Hamilton[1984] は、遠隔手続き呼び出しは非透過な RPC 機構であるため最初からローカル手続きとは異なるものとして扱うべきであると言っている。同様に、Argus の RPC の設計者 [Liskov and Scheifler 1983] は、RPC システムはユーザに対してメッセージパッシングの下位レベルの詳細は隠蔽するべきであるが、長い遅延は呼び出し側には隠蔽すべきでないとの意見を持っている。すなわち、呼び出し側はアプリケーション固有に障害や遅延の処理ができる柔軟性を持つべきである。結論として、共通的な環境におけるセマンティクス透過性は不可能であるが、分散アプリケーションプログラマに使い易さを提供できればそれで十分である。

4.4 RPC の実現方法

セマンティクス透過性の目標を達成するために、RPC 機構はスタッブ **(stubs)** の概念に基づいて実現され、このスタッブはプログラマに対して下位の RPC システムインタフェースを隠蔽することによって完全に通常 (ローカル) の手続き呼び出しの方法を提供する。RPC はクライアントプロセスとサー

バプロセスを含む。したがって、クライアントとサーバプロセスに対して下位の RPC システムのインタフェースを隠蔽するために、2 つのプロセスに対してそれぞれのスタッブがある。さらに、ネットワークやその機能の詳細を隠蔽するために、RPC 通信パッケージ (RPC 実行時ルーチンと呼ばれる) はクライアントとサーバの両側に使用される。したがって、RPC 機構の実現は、次の 5 つのプログラム要素を含む [Birrell and Nelson 1984]。

1. クライアント
2. クライアントスタッブ
3. RPC 実行時ルーチン
4. サーバスタッブ
5. サーバ

これらの相互作用を図 4.2 に示す。クライアントとクライアントスタッブ及び RPC 実行時ルーチンがクライアントマシンで動作し、サーバとサーバスタッブされに RPC 実行時ルーチンがサーバマシンで動作する。これらの要素の動作について次に説明する。

4.4.1　クライアント

クライアントは、遠隔手続き呼び出しを起動するユーザプロセスである。遠隔手続き呼び出しを行うために、クライアントはクライアントスタッブ内に存在する手続きを呼び出すために通常の手続き呼び出しを行う。

4.4.2　クライアントスタッブ

クライアントスタッブは次の 2 つの処理を行う。

■クライアントから呼び出し要求を受け取ると、目的の手続きの仕様に合わせて、パラメータをメッセージに設定してサーバスタッブに送るようにローカルの RPC 実行時ルーチンに依頼する。

■手続き実行の結果を受けると、結果を取り出し、クライアントに渡す。

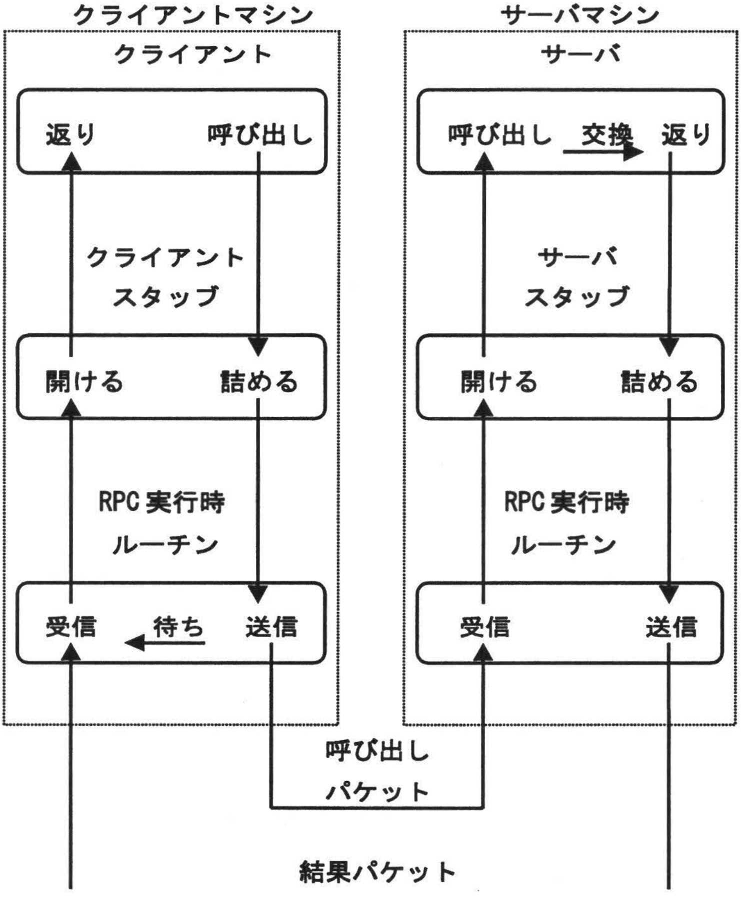


図 **4.2** RPC 機構の実装方法

4.4.3　RPC 実行時ルーチン

RPC 実行時ルーチンは、クライアントとサーバ間のネットワーク上のメッセージ転送の処理を行う。これは、再送、確認通知、パケット中継及び暗号化に関する処理を行う。クライアントマシンの RPC 実行時ルーチンは、クライアントスタッブから呼び出し要求メッセージを受け取ると、それをサーバマシンに送る。サーバマシンから手続き実行の結果を受けると、クライアントマシンにそれを転送する。クライアントマシンから呼び出し要求メッセージを受けるとサーバマシンはそれを渡す。

4.4.4　サーバスタッブ

サーバスタッブの処理は、クライアントスタッブの処理に似ている。次の 2 つの処理を行う。

■ローカル RPC 実行時ルーチンから呼び出し要求メッセージを受けると、サーバスタッブはサーバ内の適切な手続きを通常の呼び出しによって呼び出す。

■サーバから手続き実行の結果を受けると、サーバスタッブはメッセージに結果を設定し、クライアントスタッブにそのメッセージを送信するように RPC 実行時ルーチンに依頼する。

4.4.5　サーバ

サーバは、サーバスタッブから呼び出し要求を受信すると、手続きを実行し、その結果をサーバスタッブに返す。

この方式の全体的に良い点は、ローカルのカーネルに代わって遠隔で処理が行われるためクライアントの処理が軽減されることである。クライアントに手続き呼び出し後、制御が返されると、手続き実行の結果が返されている。したがって、クライアントに限ると、遠隔のサービスは従来 (ローカル) の手続き呼び出しによって利用でき、第 3 章で説明した send プリミティブや receive プリミティブを必要としない。メッセージパッシングの詳細は全てクライアントスタッブとサーバスタッブによって隠蔽され、メッセージパッシングに必要な処理についてクライアントとサーバは関与しなくてよい。

4.5 スタッブの生成

スタッブは、次の2つの方法で生成される。

1. **手動 (manually)**:この方法では、RPCの実装者はユーザがスタッブを構成するための変換機能を提供する。この方法は実現が単純であり、大変複雑なパラメータタイプも処理できる。

2. **自動 (automatically)**:この方法は、最も共通的に使用される。これは、クライアントとサーバ間のインタフェースを定義するための**インタフェース定義言語 (Interface Definition Language)(IDL)**を使用する。インタフェース定義は、主にインタフェースでサポートされる手続き名の一覧とそのパラメータとその結果のタイプによって行われる。これは、クライアントやサーバにとってコンパイル時の型検査を独立に行う場合や呼び出しシーケンスを生成するために有効な情報である。しかしながら、インタフェース定義はRPCがデータ領域やネットワークに転送するデータを減少するための情報を含む。例えば、インタフェース定義は各パラメータが入力か出力か、若しくは両方かを示し、入力であればクライアントからサーバにコピーされ、出力であればサーバからクライアントにコピーされる。同様に、インタフェース定義は、RPC呼び出しによりデータを処理するための型定義、列挙された型及び定義された定数についての情報を含む。このような情報がクライアントとサーバ両者に必要でなければ別々に置かれる。(インタフェース定義の例は図4.21を参照)

サーバのインタフェースの手続きのプログラムはインタフェースの**エクスポート (export)**と呼ばれ、クライアントのインタフェースから手続きを呼び出すプログラムをインタフェースの**インポート (import)**と呼ばれる。分散アプリケーションを書く場合、プログラマは最初にIDLによってインタフェース定義を書く。その後、インタフェースのインポートを行うクライアントプログラムとインタフェースのエクスポートを行うサーバプログラムを書く。インタフェース定義はIDLコンパイラによって処理され、クライアントとサーバの組み合わせで構成される既存のコンパイラが処理できるようなものを生成する。特に、インタフェース定義から、IDLコンパイラは各手続きに対してクライアントスタッブ手続きとサーバスタッブ手続き、及び各スタッブ手続き内の適切な結集動作や非結集動作(本章で説明する)、さらにインタフェース定義のデー

タタイプをサポートするためのヘッダファイルを生成する。ヘッダファイルは、クライアントとサーバプログラムの両方のソースプログラムに含まれる。クライアントスタッブ手続きはコンパイルされ、クライアントプログラムにリンクされる。同じようにサーバスタッブ手続きはコンパイルされ、サーバプログラムにリンクされる。IDL コンパイラは、他の言語によるインタフェース定義の処理を行うために設計することもでき、異なった言語で書かれたクライアントとサーバが遠隔手続き呼び出しによって通信することが可能である。

4.6　RPC メッセージ

いかなる遠隔手続き呼び出しも恐らく異なるコンピュータ上にあるクライアントプロセスとサーバプロセスを含む。クライアントとサーバ間の相互作用のモードは、クライアントがサーバに遠隔手続きの実行を依頼し、サーバが手続きの実行結果をクライアントに返すことである。このような相互作用のモードに基づく RPC システムの実現には次の 2 つのメッセージが使用される。

1. 特定の遠隔手続きの実行を要求するためにクライアントからサーバに送信される**呼び出しメッセージ (call message)**。

2. 遠隔手続き実行の結果を返送するためにサーバからクライアントに送信される**応答メッセージ (reply message)**。

RPC システムのプロトコルは、これらの 2 つのメッセージの形式を定義する。通常、RPC プロトコルはトランスポートプロトコルとは独立である。すなわち、RPC はメッセージがあるプロセスから他のプロセスにどのように転送されるかについて何ら関与しない。したがって、RPC プロトコルはこの 2 つのメッセージの仕様と解釈についてのみ関わる。

4.6.1　呼び出しメッセージ

呼び出しメッセージは特定の遠隔手続きの実行を要求するために使用されるため、このメッセージには、次の 2 つの基本構成要素が必要となる。

1. 実行される遠隔手続きの識別情報

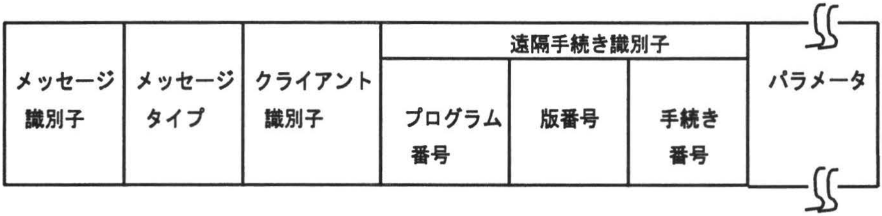


図 **4.3** 典型的な RPC 呼び出しメッセージ形式

2. 手続きの実行に必要となるパラメータ

これらの 2 つのフィールドに加えて、通常、呼び出しメッセージは次のフィールドを持つ。

3. 順序番号を含むメッセージ識別フィールド。このフィールドは次の 2 つで有効である。システム障害におけるメッセージ紛失と重複時の識別及び応答メッセージと呼び出しメッセージとの対応、特に、応答メッセージが順番に届かない場合に有効になる。

4. 呼び出しメッセージと応答メッセージを区別するためのメッセージタイプフィールド。例えば、RPC システムで、呼び出しメッセージのこのフィールドは 0 が、応答メッセージのフィールドには 1 が設定される。

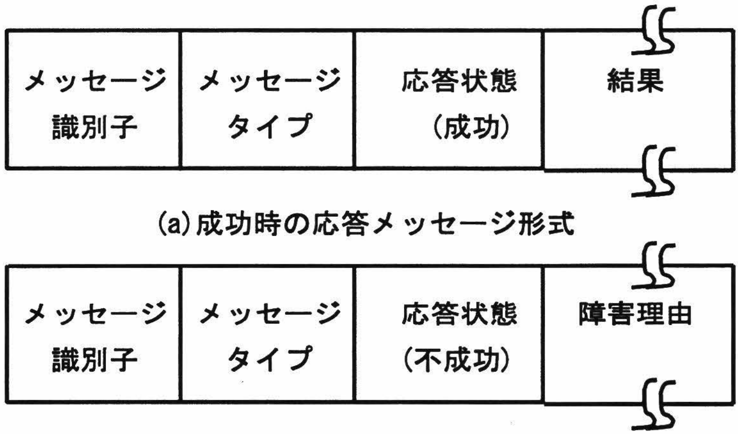
5. クライアント識別フィールド。これは、RPC のサーバが応答メッセージをどのクライアントに返すかを識別するためと、サーバが手続きの実行を行う上でクライアントプロセスを認証するための 2 つの目的に使用される。

典型的な RPC 呼び出しメッセージの形式を図 4.3 に示す。

4.6.2 応答メッセージ

RPC のサーバがクライアントから呼び出しメッセージを受信すると、次のような状態が考えられる。下記に示すリストでは、サーバは何ら問題が生じていないものとする。

1. サーバは呼び出しメッセージが解釈できない。これは呼び出しメッセージが RPC プロトコルに違反している時に生じる。この場合、サーバはこのような呼び出しを拒否する。



(b) 不成功時の応答メッセージ形式

図 4.4　典型的な RPC 応答メッセージ形式

2. サーバはクライアント識別情報からクライアントがこのサービスを使用できないことを検出する。サーバは手続きの実行を行わないで、不成功の応答を返す。

3. サーバは呼び出しメッセージの遠隔手続き識別フィールドの遠隔プログラム、版若しくは手続き番号が使用できない。この場合も、サーバは手続きの実行を行わないで不成功の応答メッセージを返す。

4. この時点で呼び出しメッセージに指定された遠隔手続きの実行に移る。したがって、遠隔手続きがパラメータを解釈できないことが発生するかもしれない。これは、クライアントとサーバのインタフェースに互換性がない場合に起きる。

5. 指定された遠隔手続きの実行中に異常状態 (例えば、0 で除算) が発生する。

6. 指定された遠隔手続きの実行は成功した。

確かに、上記の 5 つの場合では要求を処理する上での問題についての理由を付けて応答メッセージがクライアントに返され、最後の 6 番目の場合にのみ実行結果とともに成功の応答メッセージが返される。したがって、成功の応答メッセージと不成功の応答メッセージの形式は多少異なる。これらの典型的な RPC の応答メッセージの形式を図 4.4 に示す。

応答メッセージのメッセージ識別フィールドは、呼び出しメッセージと同じ

であり、呼び出しメッセージとの対応がとられる。メッセージタイプフィールドはこのメッセージが応答メッセージであることを示す。成功の応答では、応答状態フィールドは一般に0に設定され、その後に実行結果を含むフィールドが続く。不成功の応答メッセージでは、応答状態フィールドは1か誤りを示すための0以外の値が設定される。後者の場合、応答状態の値は誤りのタイプを示す。しかしながら、どちらの場合においても応答状態フィールドの後に障害の理由を記述するための特別なフィールドが置かれる。

RPCプロトコルは一般にトランスポートプロトコルとは独立しているため、RPCプロトコルの設計者が呼び出し及び応答メッセージの最大長を決めることは不可能である。したがって、トランスポートのグループに関連する分散アプリケーションでは、分散アプリケーション開発者はトランスポートのグループが指定した最大長をRPCの呼び出し及び応答メッセージが越えないように保証することは重要である。

4.7 パラメータと結果の結集

遠隔手続き呼び出しの実現において、クライアントプロセスからサーバプロセスへのパラメータの引き渡しとサーバプロセスからクライアントプロセスへの結果の引き渡しがある。これらのパラメータや結果は、基本的に言語レベルのデータ構造(プログラムオブジェクト)であり、呼び出しに含まれる2つのコンピュータ間のメッセージデータの形式で転送される。前章で、2つのコンピュータ間でメッセージデータを転送するためにはメッセージデータの符号化と復号化が必要になることを説明した。RPCにとって、このような動作は**結集 (marshaling)** と呼ばれ、基本的に次の動作を含む。

1. 遠隔プロセスに送るためのメッセージを構成する(クライアントプロセスの)パラメータと(サーバプロセスの)結果を取り出す。
2. 送信側のコンピュータで上記1をメッセージデータを符号化する。この符号化の処理ではプログラムオブジェクトから転送に適したストリーム形式に変換し、メッセージバッファに置く。
3. 受信側のコンピュータでメッセージデータを復号する。この復号の処理には受信したストリーム形式のメッセージデータからプログラムオブジェクトへの再構成が含まれる。

RPC メッセージの符号化と復号を正しく行うためには、パラメータや結果を結集するために使用する順番と表示方式 (タグ付き又は非タグ付き) が RPC のクライアントとサーバ両者に周知されていなければならない。これは、クライアントがサーバに同じインタフェース定義を使用しなければサーバはクライアントからの呼び出しを受け取れないため、クライアントとサーバ間の**型安全性 (type safety)** の程度を提供するものである。型安全性は、不正な呼び出し要求を受け付ける可能性があるためサーバに対しては格別重要である。

結集プロセスは、関連する言語で使用されるプログラミングオブジェクトの全ての型の構造を反映しなければならない。これらには、プリミティブ型、構造型及びユーザ定義の型を含む。結集手続きは、次の 2 つのグループに分類できる。

1. RPC ソフトウェアの一部分として提供されるもの。通常、スカラーデータ型の結集手続きは、スカラーデータ型から構成される複合型を結集するための手続きとともにこのグループに入る。
2. RPC システムのユーザによって定義されたもの。このグループは、ユーザ定義のデータ型とポインタを含むデータ型に対する結集手続きを含む。例えば、Cambridge 分散コンピュータシステム [Bacon and Hamilton 1987] のために開発された並行性の CLU で、ユーザ定義の型に対して型定義は結集のための手続きを含まなければならない。

良い RPC システムは、全ての遠隔手続き呼び出しに対してインラインの結集コードを生成すべきであり、ユーザは自分自身で結集手続きを書くわずらわしさから解放される。しかしながら、特に、全ての可能なデータ型を処理できるために受け入れ不可能な大規模なコードを生成するためこの目標を達成することは困難である。

4.8　サーバ管理

RPC に基づくアプリケーションにおいて、サーバ管理として考慮しなければならない 2 つの重要な課題は、サーバの実現方法とサーバの生成である。

4.8.1 サーバの実現方法

実現する形態によって、サーバは次の2つのタイプに、ステートフルとステートレスがある。

ステートフルサーバ

ステートフルサーバ (stateful server) は、ある遠隔手続き呼び出しから次の呼び出しまでクライアントの状態情報を管理する。すなわち、ステートフルサーバにクライアントが連続する2つの呼び出しを行う場合、最初の呼び出し実行結果としてクライアントに対して行われたサービスに関する状態情報は、サーバプロセスによって保持される。これらのクライアントの状態情報は、2番目の呼び出しの実行時に引き続いて使用される。

例えば、サーバで次のような動作を行うバイトストリームのファイルについて考えよう。

Open(filename, mode): この動作は指定された mode で filename で指定されたファイルのオープンを行う。サーバがこの動作を行うと、全てのオープンされているファイルのファイル状態情報を管理するために使用される**ファイルテーブル (file-table)** にこのファイルのエントリを生成する。ファイル状態情報は、通常、ファイルの識別子、オープンモード及び**読み書きポインタ (read-write pointer)** と呼ばれる正の整数のポインタの現在の位置から構成される。ファイルがオープンされると、その読み書きポインタが0に設定され、サーバはクライアントにファイルの識別子 (fid) を返し、この情報をクライアントは以降のこのファイルへのアクセスに使用する。

Read(fid, n, buffer): この動作は fid で指定されたファイルから n バイトのデータを buffer の名前を持つバッファに読み出す。サーバがこの動作を実行すると、読み書きポインタで示されたアドレスのファイルデータから n バイトをクライアントに返し、読み書きポインタを n 加算する。

Write(fid, n, buffer): この動作が実行されると、サーバは指定された buffer から n バイトのデータを取り、fid で指定されたファイルの読み書きポインタで示されたアドレスの位置にデータを書き、読み書きポインタを n 加算する。

Seek(fid, position): この動作は fid で指定されたファイルの読み書きポインタの値を position で指定した新たな値に変更する。

Close(fid): これは fid で指定されたファイルの状態情報をファイルテーブルから削除する。

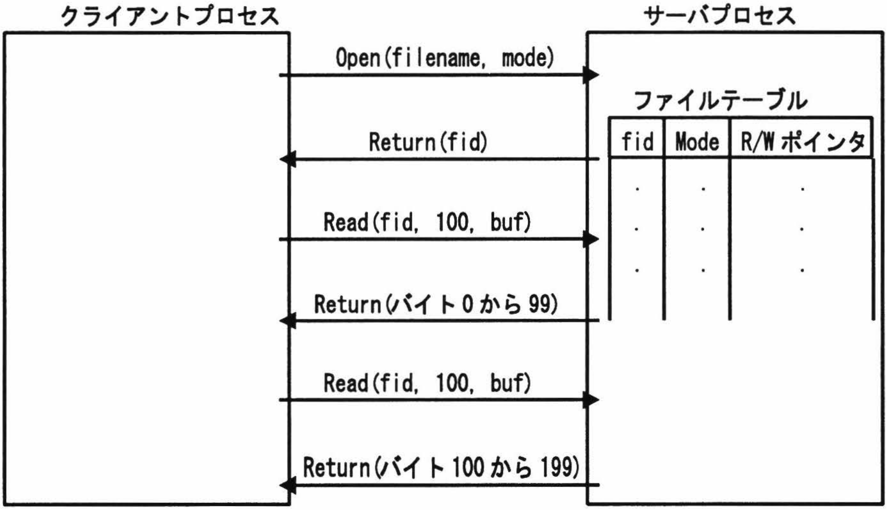


図 4.5　状態保持のファイルサーバの例

上述したファイルサーバは、クライアントがオープンしたファイルに関する状態情報を管理することからステートフルである。したがって、図 4.5 に示すようにファイルをオープンし、クライアントが 2 回 Read(fid, 100, buf) 呼び出しを行うと、最初の呼び出しで最初の 100 バイト (バイト 0 から 99) を度目の呼び出しで次の 100 バイト (バイト 100 から 199) を返す。

ステートレスサーバ

ステートレスサーバ (stateless server) は、クライアントの状態情報を管理しない。したがって、クライアントからの全ての要求は、要求する動作を正しく行うために必要な全てのパラメータを持たなければならない。例えば、サーバはバイトストリームのファイルに対する次のような動作はステートレスである。

Read(filename, position, n, buffer): この動作の実行では、サーバは filename で指定したファイルの n バイトのデータをクライアントに渡す。渡されたデータは buffer で指定されたバッファに置かれる。読まれた実際のバイト数はクライアントに通知される。読まれるファイルの位置は position パラメータで与えられる。

Write(filename, position, n, buffer): サーバがこの動作を実行すると、指定された buffer から n バイトのデータが取り出され、filename で指定されたファ

イルに書かれる。position パラメータによってファイルに書き込まれる位置が与えられる。サーバは実際に書いたバイト数をクライアントに通知する。

図 4.6 に示すように、このファイルサーバは、実行した動作に関してファイル状態情報を保持しない。したがって、クライアントが図 4.5 と同じような効果を期待するのであれば、次のような 2 つの Read 動作を行わなければならない。

Read(filename, 0, 100, buf)

Read(filename, 100, 100, buf)

この場合、クライアントは自分自身でファイルの状態情報を管理しなければならない。

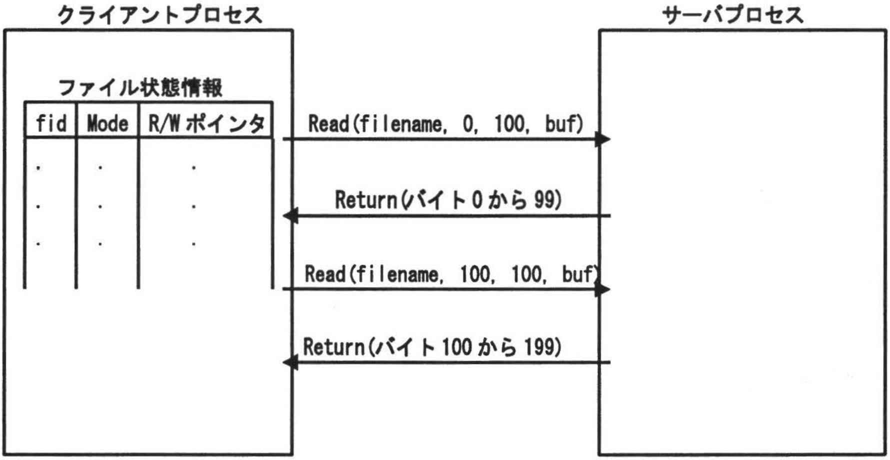


図 4.6　ステートレスのファイルサーバの例

なぜステートレスサーバなのか?

ステートフルとステートレスサーバの説明から、ステートフルサーバはクライアントに対して状態情報の管理の必要正をなくするため、このステートフルサーバは容易なプログラミングが可能になると考えられる。さらに、一般的にステートフルサーバはステートレスサーバよりも効果的である。では、この疑問、なぜステートレスサーバなのか。

多くの分散アプリケーションにおいてステートレスサーバが使用されるの

は、ステートレスサーバが障害においてステートフルサーバよりも何らかの利点を持っているためである。例えば、ステートフルサーバで、サーバが障害を起こして再開すると、それまで持っていた状態情報は失われ、クライアントは障害に気が付かないと一貫性のない結果を招く。同様に、クライアントが障害を起こして再開すると、サーバはもはや正しくない状態情報を持ち、容易に回復することができない。したがって、ステートフルサーバのクライアントは、サーバの障害を検出できるように設計されなければならなく、誤り制御処理を行う必要がある。一方、状態不情報サーバでは、クライアントはサーバからの応答が来るまで要求を再送するだけである。クライアントにとってサーバが障害なのかネットワークが障害なのかを意識することはない。したがって、再試行するように考えられているステートレスサーバは障害に対する回復処理が容易になる。

ステートフルとステートレスサーバ両者ともお互いに長所と短所を持っている。これらのサーバの選択は、まさにアプリケーションに依存している。したがって、分散アプリケーションシステム設計者は、選択に先立ってアプリケーションにとってどちらが有効かを注意深く確かめることが必要である。

4.8.2　サーバ生成セマンティクス

RPC において、クライアントプロセスからの遠隔手続き呼び出しとして実行される手続きは、クライアントプロセスとは独立したサーバプロセスに置かれる。ここでの独立性とは、クライアントとサーバプロセスがそれぞれ別の生存期間を持ち、通常別のマシン上で実行し、それぞれのアドレス空間を持っていることを意味している。サーバプロセスは遠隔手続きを呼び出すクライアントとは独立しているため、サーバプロセスはクライアントプロセス以前に生成されて実装されるか、要求時によって生成される。RPC サーバの生存期間に基づいて、**呼び出しインスタンスサーバ (instance-per-call server)**、**トランザクション／セッションインスタンスサーバ (instance-per-transaction /session server)** 若しくは**永続サーバ (persistent server)** に分類される。

呼び出しインスタンスサーバ

この分類に属するサーバは、単一の呼び出し期間のみ存在する。このタイプのサーバは、呼び出しメッセージが到着した時のみサーバマシンの RPC 実行時ルーチンによって生成される。このサーバは、呼び出しの実行が終了すると

消滅する。

サーバ生成に対するこの方法は、次のような問題があるため一般には使用されない。

■このタイプのサーバは呼び出しが終了すると消滅するためにステートレスである。したがって、サーバ呼び出しにまたがって保持すべき状態情報は、クライアントプロセスかサポートしているオペレーティングシステムによって注意深く扱われなければならない。呼び出し間の状態情報がオペレーティングシステムに関連する場合、遠隔手続き呼び出しは高価なものになる。一方、呼び出し間の状態情報がクライアントプロセスによって管理される場合には、状態情報を各呼び出し毎にサーバとやり取りしなければならない。これは、クライアントサーが間のインタフェースのデータ方式を低下させ、プログラマに対して RPC 機構の優位性を低下させる。

■分散アプリケーションが連続的にサーバの処理を呼び出したい場合、資源(バッファや制御構造のためのメモリ空間) 割り当てと解放が繰り返し起こるため、この方式では高価になってしまう。したがって、サーバの生成や削除に関わるオーバヘッドが遠隔手続き呼び出しのコストにはね返る。

セッションインスタンスサーバ

この分類に属するサーバは、クライアントとサーバが相互作用のためのセッションを持っている間存在する。このタイプのサーバはセッションの間存在するため、呼び出し間の状態情報を保持され、多くの呼び出しが含まれるクライアントサーバのセッションでのサーバ生成と削除に関するオーバヘッドは最小化される。

この方法では、一般に各サービスに対してサーバ管理者がいる。全ての管理者は結合エージェント (クライアントとサーバを結合するための結合エージェント機構については本章の後半で述べる) とともに登録される。クライアントが結合エージェントに問い合わせし、必要なサービスのタイプを指定すると、結合エージェントはこのタイプのサーバ管理者のアドレスをクライアントに返す。クライアントは、サーバを生成するためにこのサーバ管理者に問い合わせる。このサーバは要求するクライアントによって排他的に使用され、クライアントがサーバが必要でなくなったことをサーバ管理者に通知すると消滅する。

このタイプのサーバは、呼び出し間の有効な状態情報を保持できるため、クライアントに明解でより概念的なインタフェースを提供することができる。こ

のタイプのサーバは、単一のクライアントのみサービスするために 1 つの状態情報のみを管理する。

永続サーバ

永続サーバは、一般に限りなく存在する。上記の 2 つのサーバは、要求にしたがって排他的に生成されるため複数のクライアントから共有できない。このサーバは、これらとは異なり、多くのクライアントによって共有される。

このタイプのサーバは、クライアントが使用開始する前に生成され実装される。各サーバは、結合エージェントとともに登録され、独立にサービスを提供する。クライアントが特定のサービスタイプに対して結合エージェントに問い合わせると、結合エージェントはある優先順位 (現在サービスしているクライアントの数が少ない) などにより対応するサーバを選択し、そのサーバのアドレスをクライアントに通知する。

永続サーバは、同時に複数のクライアントに対応する。この場合、サーバは多くのクライアントからの要求を処理し、複数の状態情報を並行的に管理する。永続サーバが複数のクライアントに共有される場合、遠隔手続きはそのように設計されていなければならないし、異なるクライアントからの並行的な要求は相互に関連させてはいけない。

永続サーバは、全体の性能やシステムの信頼性の向上に有効である。このため、負荷分散や障害対応の観点から同じサービスタイプの永続サーバを異なるマシンに複数置くことが考えられる。

4.9　パラメータパッシングセマンティクス

パラメータパッシングセマンティクスの選択は、RPC 機構を設計する上で重要である。この選択には、値パラメータと参照パラメータがある。

4.9.1　値パラメータ

値パラメータ (**call-by-value**) 方式では、全てのパラメータはメッセージにコピーされ、クライアントからサーバにネットワークを介して転送される。これは、簡単な型、例えば整数、カウンタ、小さい配列などでは問題がない。しかしながら、多次元配列、木構造などのより大きな型を引き渡すことは、多くの転送時間を使ってしまう。したがって、この方式は大容量なデータを含むパ

ラメータを引き渡すためには適していない。

値によって大容量のパラメータを引き渡すことの良い点は、遠隔手続き呼び出しが高価になってしまうことをユーザに意識させることである。言い替えれば、ユーザは不必要なデータの引き渡しを最少にするためにクライアントとサーバ間のインタフェース設計を十分に検討しなければならない。したがって、RPC パラメータパッシングセマンティクスを選択する前に、クライアントサーバ間のインタフェース設計を十分に行い、転送するパラメータが最小になるように決めることが重要である。

4.9.2 参照パラメータ

多くの RPC 機構は、クライアントとサーバが異なるアドレス空間、恐らく別のマシンに存在し、ポインタや**参照 (reference)** によるパラメータの引き渡しは意味がなくなるために、パラメータの引き渡しに値パラメータセマンティクスを使用する。

しかしながら、2、3 の RPC 機構はパラメータへのポインタをクライアントからサーバに渡す、参照によるパラメータの引き渡し方法を行う。このようなものは、一般にシステムの全てのプロセスが同じアドレス空間を共有するような閉じたシステムである。例えば、分散共有メモリ機構 (第 5 章で述べる) を持つ分散システムは参照パラメータによる引き渡しを行う。

オブジェクトの呼び出しのために RPC 機構を使用するオブジェクトに基づくシステムでは、**参照パラメータ (call-by-reference)** は**オブジェクト参照パラメータ (call-by-object-reference)** と呼ばれる。これは、オブジェクトに基づくシステムでは、変数の値はオブジェクトへの参照であり、呼び出し時に渡される参照 (オブジェクト名) となる。

Emerald[Black et al. 1986、1987] の設計者は、遠隔手続き呼び出しでの遠隔動作でのパラメータへのアクセスは付加的な遠隔呼び出しをもたらすために、分散システムにおけるオブジェクト参照パラメータの使用は性能問題があると考えている。したがって、このような多くの遠隔参照を避けるために、Emerald では**移動パラメータ (call-by-move)** と呼ぶ新しいパラメータパッシングモードを採用している。この移動パラメータでは、パラメータはオブジェクト参照パラメータの方式と同じように参照として渡されるが、呼び出し時、パラメータオブジェクトは宛先ノード (呼ばれる側) に移される。呼び出し後は、パラメータオブジェクトは呼び出し側のノード又はそのまま呼ばれた側の

ノードに置かれる(それぞれ、滞在パラメータ **(call-by-visit)**、移動パラメータ **(call-by-move)** と呼ばれる)。

明らかに、パラメータパッシングにおいて移動パラメータモードを使用することは、ノード間を移動できるモバイルオブジェクトをサポートするシステムが必要となる。Emerald のオブジェクトは移動可能である。

移動パラメータは、オブジェクト参照パラメータと同様パラメータパッシングセマンティクスを変えるものではない。したがって、移動パラメータは、基本的に便利であり、性能を向上させる。これは、次の 2 つの動作に起因するためである。

■まず、各移動パラメータのオブジェクトを呼び出される側のノードに移動する。
■オブジェクトを呼び出す。

しかしながら、移動を別々に行うことはネットワークに複数のメッセージの転送を行うことになってしまう。そのため、移動パラメータによるパラメータパッシングでは、呼び出すメッセージと同じネットワークパケットにパラメータオブジェクトを挿入し、ネットワークのトラヒックとメッセージ数を減少することができる。

移動パラメータは、呼ばれる側の参照するコストを軽減するが、呼び出しそのもののコストは高くなる。もし、パラメータオブジェクトが変わる又は共有される場合、呼び出し側からの参照コストを高めてしまう [Black et al. 1987]。

4.10　呼び出しセマンティクス

RPC では、呼び出しと呼ばれるプロセスは恐らく異なるノードに置かれる。このため、呼び出しノードと呼ばれるノードが別々に障害になり、その後再開することがある。さらに、両者の間の通信リンクの障害も起こり得る。したがって、RPC の機能は下記の理由から異常な動作を起こす。

■呼び出しメッセージの紛失
■応答メッセージの紛失
■呼ばれるノードの障害による再開
■呼び出しノードの障害による再開

RPCでの呼び出しノードの構成要素は、これらの障害に対応するための処理を含まなければならない。明らかに、呼び出し側の手続きはこれらの障害の詳細に関係しない。したがって、障害処理は、一般にRPC実行時ルーティングに含まれる。障害時の遠隔手続きがどのように実行されるかを決めるRPCシステムの呼び出しセマンティクスは、このRPCルーティングの処理内容に依存する。この内容は、RPCシステムとしてアプリケーションプログラマが可能な呼び出しセマンティクスから柔軟に選択できるように設計されなければならない。RPCシステムでの各種の呼び出しセマンティクスを次に説明する。

4.10.1 不確実呼び出しセマンティクス

この**不確実呼び出しセマンティクス (possibly or may-be-call semantics)**は、最も弱いセマンティクスで実際のRPCには適用できないが、ここでは説明を行う。この方式は、呼び出し側が呼ばれた側からの無制限の応答待ちを防ぐためにタイムアウトの機構を使用する。すなわち、呼び出し側はあらかじめ決められたタイムアウト時間まで待ち、実行を継続する。したがって、セマンティクスとして呼ばれた側からの呼べ出しメッセージの受理や手続きの実行について必ずしも保証していない。このセマンティクスは、応答メッセージが重要でないアプリケーションやメッセージの転送誤りが極めて少ないLANでのアプリケーションに限られる。

4.10.2 最後1回呼び出しセマンティクス

この呼び出しセマンティクスは、3.9節で説明し、図3.10で示したものに類似している。これは、応答メッセージが受信されるまでタイムアウトにより呼び出しメッセージを再送する考え方である。すなわち、呼び出し側による遠隔手続きの呼び出し、呼ばれた側による手続きの実行、呼び出し側への結果の返送が、正しく呼び出し側に結果が届くまで繰り返される。確かに、以前の(捨てられた)呼び出しが障害などと関係して副作用をもたらすかもしれないが、最後に実行した呼び出しの結果が呼び出し側によって使用される。ここではこのセマンティクスを**最後1回セマンティクス (last-one call semantics)**と呼ぶ。

この最後1回セマンティクスは、RPCに2つのプロセッサ(ノード)のみが含まれるのであれば、上で述べたように容易に実現できる。しかしながら、障

害を伴う場合の最後 1 回セマンティクスを達成することは、2 つのプロセッサ (ノード) を越える結合された RPC では予期せぬ動作が生じることがある [Bal et al. 1989]。例えば、ノード N1 のプロセス P1 がノード N2 の手続き F1 を呼び出し、これがノード N3 の手続き F2 を再度呼び出す場合を考える。N3 のプロセスは F2 を実行中に、ノード N1 が障害を起こす。ノード N1 のプロセス再開し、P1 から F1 への呼び出しが繰り返される。F1 の 2 度目の呼び出しにより、ノード N3 の手続き F2 の呼び出しを再度行う。しかしながら、ノード N3 はノード N1 の障害を知らない。したがった、手続き F2 はノード N3 上で 2 回実行され、N3 は F2 の実行結果を返すが、恐らく最後 1 回セマンティクスは正しくないであろう。

このような場合の最後 1 回セマンティクスを達成する上での基本的な困難性は、孤児の呼び出しによって生じる。**孤児呼び出し (orphan call)** は、呼び出した親 (呼び出し側) が障害によって消滅したもの呼び出しのことである。最後 1 回セマンティクスを達成するためには、障害を起こしたプロセスを再開する前に孤児呼び出しを終了しなければならない。これは、一般に、終了するのを待つか、強制的に終了させるか (**孤児根絶 (orphan extermination)**) である。この動作は容易ではないため、他のより弱いセマンティクスが RPC として提案されている。

4.10.3　最後呼び出しセマンティクス

最後呼び出しセマンティクス (last-of-many call semantics) は、孤児呼び出しを無視すること [Bal et al. 1989] を除けば、最後 1 回セマンティクスと同じである。孤児呼び出しを無視する簡単な方法は、各呼び出しにユニークな呼び出し識別子を付与することである。呼び出しが繰り返されると、新たな呼び出し識別子を与える。各応答メッセージは、対応する呼び出し識別子を持つ。呼び出し側は、最後に呼び出し時の識別子と一致する応答メッセージのみを受信し、その他の応答メッセージは廃棄する。

4.10.4　最低 1 回呼び出しセマンティクス

最低 1 回呼び出しセマンティクス (at-least-once call semantics) は、最後呼び出しセマンティクスよりさらに弱い呼び出しセマンティクスである。これは、呼び出しが 1 回若しくは複数回行われても、どの結果かを示さないもの

である。孤児呼び出しの対応をしないでタイムアウトによる再送であるため実現は簡単である。すなわち、ネストされた呼び出しの場合、たとえ孤児呼び出しがあったとしても最初の応答メッセージの結果を採用し、他のものは無視するため、孤児呼び出しからの応答か否かは考慮しない。

4.10.5 ちょうど1回呼び出しセマンティクス

ちょうど1回呼び出しセマンティクス (exactly-once call semantics) は、呼び出しの再送が行われても手続きの実行は1回とするため最も強くて望ましい呼び出しセマンティクスである。最後1回、最後及び最低1回の各呼び出しセマンティクスともこれを保証できない。これらのセマンティクスの主な欠点は、手続きが同じパラメータで複数回実行された場合に同じ結果と同じ副作用を得られることを保証するための繰り返し等価のインタフェースの設計をプログラマに要求することにある。例えば、シーケンシャルファイルの固定長レコードの読み書きの例 [Wilbur and Bacarisse 1987] を考える。このファイルから連続的にレコード読むための手続きは次のようになる。

ReadNextRecord(Filename)

初期化とファイルの終了を無視すると、この手続きの実行により指定されたファイルの次のレコードを読み出す。明らかに、この手続きは、応答メッセージが失われて重複した呼び出しが再送されることを考えないとしてもこの手続きを複数回実行する時の返されるレコードが異なるために繰り返し等価ではない。これは、この手続きを実現する場合サーバはファイルをオープンした各クライアントに対して現在のレコード位置を保持することで起きる。したがって、ファイルから次のレコードを読むための繰り返し等価インタフェースを設計するためには、各クライアントが現在のレコード位置を保持し、サーバはステートレスであることが重要であり、すなわち、クライアントの状態をサーバに持たないことである。この考え方に基づいたシーケンシャルファイルの次のレコード読み出しのための繰り返し等価な手続きは、次のようになる。

ReadRecordN(Filename, N)

これは、指定されたファイルのN番目のレコードを返す。この場合、クライ

アントは希望するレコードを読むためにNの値を正しく指定しなければならない。

しかしながら、非繰り返し等価なインタフェースは繰り返し等価なものに容易に変換することはできない。例えば、同じシーケンシャルファイルの新しいレコードを追加する場合を考える。

AppendRecord(Filename, Record)

これは、繰り返される実行は同じレコードの内容をファイルに追加するため明らかに繰り返し等価ではない。このインタフェースを上の代わりに次の 2 つの手続きによって繰り返し等価に変換することができる。

GetLastRecordNo(Filename)

WriteRecordN(Filename, Record, N)

最初の手続きは、ファイルの現在のレコード番号を返し、2 番目はファイルの指定した場所にレコードを書き込む。さらに、レコードを追加するためにクライアントは次の 2 つの手続きを使用する。

Last = GetLastRecordNo(Filename)

WriteRecordN(Filename, Record, Last)

ちょうど 1 回セマンティクスにおいては、呼び出しセマンティクスは手続きを 1 度のみ実行するようにするため、プログラマはサーバを繰り返し等価に実現することから解放される。3.9 節や図 3.12 の例で説明したように、ちょうど 1 回呼び出しセマンティクスの実現はタイムアウト、再送、繰り返される呼び出しへの同じ呼び出し識別子の付与及び呼ばれる側での応答キャッシュによって行われる。

4.11 RPCの通信プロトコル

遠隔手続き呼び出しに基づいて開発された異なるシステムは、別々のIPCの要望を持っている。これらの異なる要望に基づいて、RPCとして幾つかの通信プロトコルが提案されている。これらのプロトコルの概要について次に説明する。

4.11.1 要求プロトコル

このプロトコルは、R(要求)プロトコルとも呼ばれる[Spector 1982]。これは、RPCにおいて呼ばれた手続きは実行結果を何ら返すこともなければ、クライアントも手続きを実行したことの確認を要求しない。このプロトコルでは確認通知や応答メッセージがなく、唯一呼び出しのメッセージが転送される(クライアントからサーバへ)(図4.7)。クライアントは、一般に要求メッセージを送ると応答メッセージを待つ必要がないため、即座に次の処理に移る。このプロトコルは、不確実呼び出しセマンティクスであり、要求メッセージの再送は行わない。

Rプロトコルを使用するRPCは、**非同期RPC (asynchronous RPC)**と呼ばれる。非同期RPCは、クライアントとして応答を要求しないような分散アプリケーションにおいてはクライアントとサーバの両者組み合わせた性能の向上を可能にする。クライアントの性能は、呼び出した後ブロックされなく即座に次の処理に移れるため向上できる。一方、サーバの性能は、要求に対する応答の生成と送信がないため良くなる。アプリケーションの1つとして分散ウィンドウシステムがある。X-11[Davison et al. 1992]のような分散ウィンドウシステムは、サーバとしてプログラムされ、クライアントは表示画面上のウィンドウに項目を表示することを要求するアプリケーションプログラムである。ウィンドウに項目を表示するために、一般にクライアントは要求に対する応答が必要とされないため応答を待つことなく、次々とサーバに多くの要求(各要求は表示情報を少し変更するために比較的少ない情報である)を送信する。

非同期RPCでは、RPC実行時ルーチンは通信障害時要求の再送の処理を行わない。このことは、RPCに信頼性のないデータグラムトランスポートプロトコルであるUDPが使用されると、要求メッセージはクライアントが知らぬ間に失われることもある。信頼性のないトランスポートプロトコルでの非同期RPCを使用するアプリケーションは、このような状況に対応できるようにし

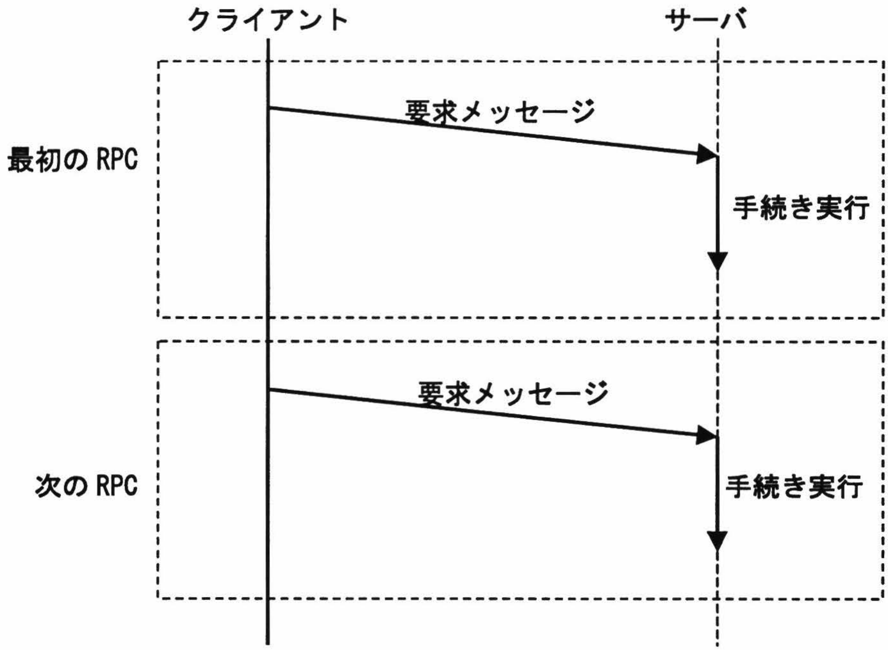


図 4.7　要求 (R) プロトコル

なければならない。しかし、信頼性のあるコネクション型トランスポートプロトコルである TCP を RPC に使用する場合には、このような場合信頼性のある送達が行われるため要求メッセージの再送に気にする必要はない。

信頼性のないトランスポートプロトコルの非同期 RPC は、一般的に周期的に更新サービスを行う場合に有効である。例えば、分散システムにおける時刻サーバノードは非同期 RPC 機能を使用して他のノードに T 秒毎に時刻同期メッセージを送信する。この場合、メッセージが失われても、正しい時刻は次のメッセージで転送される。各ノードは多くの更新メッセージを受信できなかった場合には最後の更新メッセージを受信保持しておくことで防ぐことができる。多くの更新メッセージを受信できなかったノードは、ある時間経過した後信頼性のある更新を行うために時間サーバノードに全く別の要求メッセージを送信することができる。

4.11.2 要求／応答プロトコル

このプロトコルは、RR(要求／応答) プロトコルとも呼ばれる [Spector 1982]。これは、簡単な RPC のシステムを設計する場合に有効である。**単純 RPC (simple RPC)** は、全てのパラメータと同様に全ての結果も単一のパケットバッファに入り、呼び出しの間の期間や呼び出し間の間隔が短い (クライアントとサーバ間のパケット転送時間よりも短い)[Birrell and Nelson 1984]。このプロトコルは、明示的な確認通知メッセージを削除するために暗黙の確認通知の使用する方式であり、次のようになる。

■ サーバからの応答メッセージはクライアントの要求メッセージの確認通知と見なす。

■クライアントからの引き続く呼び出しパケットはその前のサーバからの応答メッセージの確認通知と見なす。

RR プロトコルにおけるクライアントとサーバ間のメッセージの交信を図 4.8 に示す。この図では呼び出し毎に 2 つのパケット (各方向に 1 つ) があるものとしてプロトコルを表している。

基本的な RR プロトコルであるため障害処理の機能については表していない。したがって、メッセージの紛失に対してはタイムアウトと再送技術が RR プロトコルとして行われる。この技術では、あらかじめ決められた時間が経過しても応答メッセージをクライアントが受信しない場合には、クライアントは要求メッセージを再送する。明らかに、重複した要求メッセージが生じるため、この技術を用いる RR プロトコルは最低 1 回呼び出しセマンティクスである。しかしながら、サーバは重複要求メッセージの処理や再処理の必要でない要求に対する応答メッセージを再送できるための応答キャッシュに応答を保持することでちょうど 1 回呼び出しセマンティクスをサポートできる。この技術の詳細については 3.9 節で説明した。

4.11.3 要求／応答／確認通知プロトコル

このプロトコルは、**RRA (request/reply/acknowledge-reply)**(要求／応答／確認通知) プロトコルと呼ばれる [Spector 1982]。RR プロトコルによるちょうど 1 回呼び出しセマンティクスは、サーバの応答キャッシュに応答の記

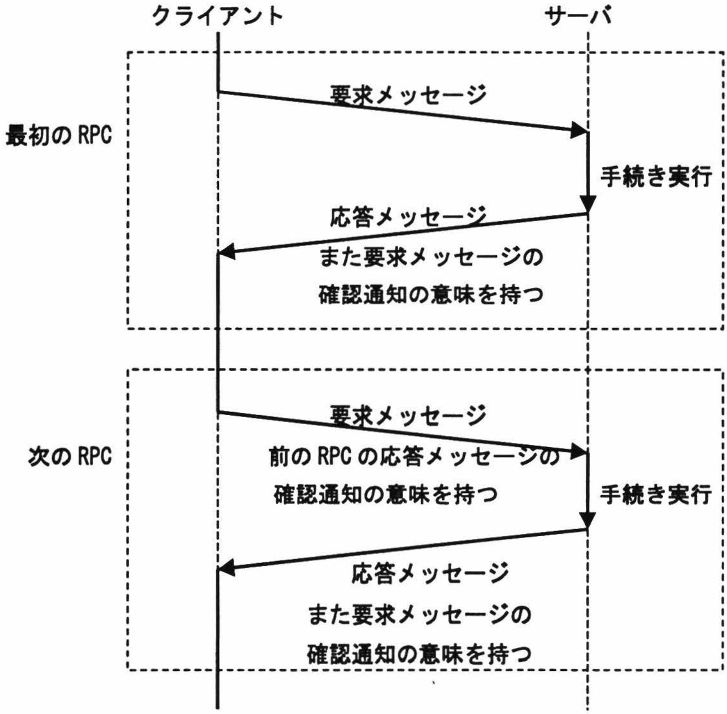


図 **4.8**　要求／応答 (RR) プロトコル

録を保持することを必要とする。サーバが多くのクライアントと通信する場合、サーバには大きな記憶容量が必要となる。幾つかの実現法では、サーバはある時間を経過したデータは廃棄することで容量を制限している。しかしながら、このような方法はクライアントに正しく応答を返せなかった場合に応答の紛失を引き起こす。この RR プロトコルの制限に対応するために、クライアントが応答メッセージを受信した場合に確認通知を返す RRA プロトコルが使用される。サーバは、クライアントから確認通知を受けると応答キャッシュからそれに関する情報を削除できる。図 4.9 に示すように、RRA プロトコルは、1 つの呼び出しに 3 つのメッセージを転送する (クライアントからサーバに 2 つ、サーバからクライアントに 1 つ)。

RRA プロトコルにおいて、確認通知メッセージそのものが紛失することがある。したがって、RRA プロトコルの実装においては、要求メッセージに関す

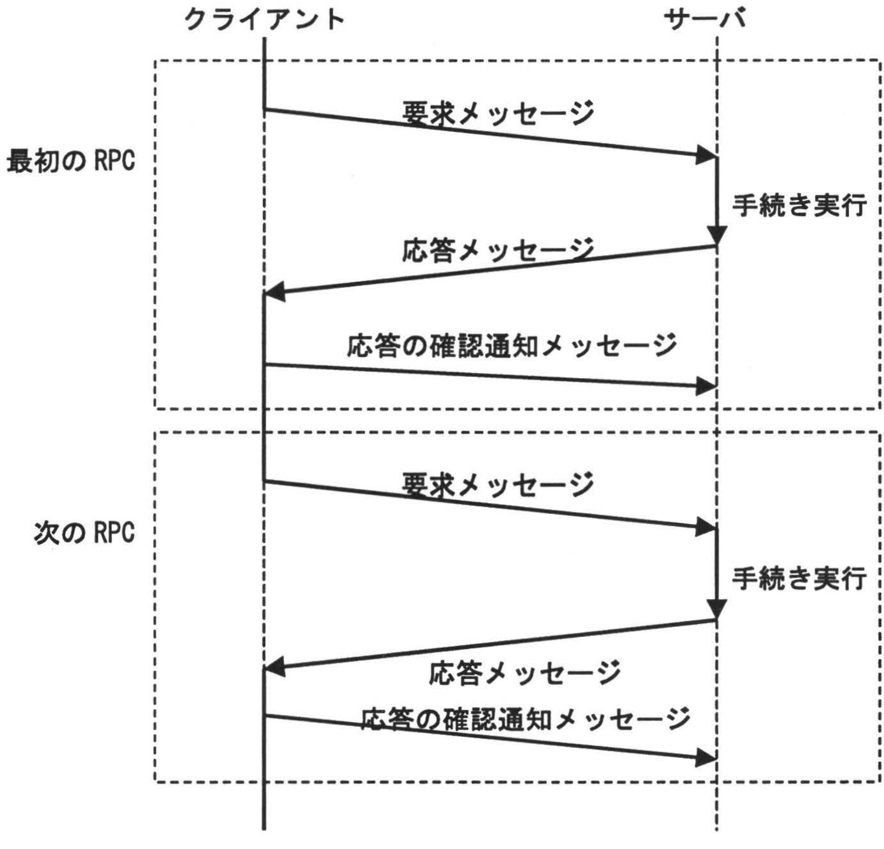


図 **4.9** 要求／応答／確認通知 (RRA) プロトコル

る順番になったユニークなメッセージ識別子が必要となる。各応答メッセージは、対応する要求メッセージのメッセージ識別子を含み、また各確認通知メッセージも同じメッセージ識別子を持つ。これによって、要求メッセージと応答メッセージの対応、応答メッセージと確認通知メッセージの対応を取ることが可能となる。クライアントは、以前の全ての要求に対して応答メッセージを受け場合に応答メッセージに対する確認通知を返す。これにより、確認通知メッセージは、その番号より若いメッセージ識別子を持つ要求メッセージに対する全ての応答の受信の確認通知として解釈される。したがって、確認通知メッセージの紛失は問題とならない。

4.12　複雑な RPC

Birrell と Nelson[1984] は、複雑な RPC を次の 2 つに分類している。

1. 長期間の呼び出しと呼び出し相互の間が長い RPC
2. 単一のデータグラムパケットに入らないパラメータや結果を持つ RPC

このような 2 つのタイプの複雑な RPC を処理するために別のプロトコルが使用される。

4.12.1　長期間の呼び出しと呼び出し相互の間が長い RPC

このタイプの RPC を処理するためには次の 2 つのどちらかの方式が使用される [Birrell and Nelson 1984]。

1. **クライアントからの定期的なサーバへの打診**:この方式では、クライアントがサーバに要求メッセージを送信すると、定期的にサーバに打診パケットを送信し、サーバはこれに確認通知を返すことが要求される。これによって、クライアントはサーバのクラッシュや通信リンクの障害を検出でき、異常状態を関連するユーザに通知できる。元の要求メッセージのメッセージ識別子は、各打診メッセージに含まれる。したがって、元のメッセージが紛失しても、このメッセージに対する打診パケットへの応答によって、サーバはクライアントに打診パケットに対応する要求メッセージは受信していないことを知らせることができる。サーバからこのような応答を受信すると、クライアントは元の要求を再送する。

2. **サーバによる定期的な確認通知の生成**:この方式では、サーバが期待される再送間隔よりも短く次のパケットを生成できない場合には、サーバは応答を自発的に生成する。したがって、長期間の呼び出しではサーバは、呼び出し期間に比例して複数の確認通知を生成する。クライアントがあらかじめ決まっているタイムアウト時間内にサーバから要求に対する応答又は確認通知を受信しない場合、サーバがクラッシュしたか通信リンクが障害になったと見なす。この場合、関連するユーザに異常状態が通知される。

4.12.2　長いメッセージを含む RPC

幾つかの RPC では、パラメータや結果は単一のデータグラムパケットに入りきらないこともある。例えば、ファイルサーバでは write 動作時のパラメータや read 動作時の結果が大量なデータとして転送される。このような RPC を処理する単純な方法は、1 つの論理的な RPC に対して幾つかの物理的な RPC を使用することである。この方法は、データの量に関係なく各 RPC のオーバヘッドを生じるため効果的ではない。

このような場合の RPC を処理する別な方法は、複数のデータグラムのメッセージを使用することである。この方法では、大きい RPC のパラメータや結果は分割され、複数のパケットとして転送される。通信性能を改善するために、複数のデータグラムのメッセージに対して 1 つの確認通知パケットが使用される。この場合、3.9 節で述べた同じ方法が複数のデータグラムの RPC メッセージの紛失や順序誤りに対して使用される。

RPC システムでは小さなサイズに制限される。例えば、サンマイクロシステムの RPC では 8 キロバイトに制限される。したがって、このようなシステムでは、制限を越えるメッセージを含む RPC は幾つかの物理的な RPC に分けて処理されなければならない。

4.13　クライアントとサーバの結合

クライアント (実際はクライアントスタッブ) は、クライアントとサーバ間で遠隔手続き呼び出しが行われる前にサーバの位置を知る必要がある。呼び出しによってクライアントとサーバを関係付けるプロセスを**結合 (binding)** と呼ぶ。アプリケーションの観点からすると、結合のモデルはサーバの「エクス

ポート」動作をサービスとして提供するために、サーバの位置について RPC 実行時ルーチンに問い合わせるクライアントの「インポート」に関連付け、相互に必要となる状態を確立することである [Bershad et al. 1987]。クライアントとサーバの結合は、幾つかの課題に対する適切な処理を含む。

1. クライアントは結合をしたいサーバをどのように指定するか？
2. 結合処理は指定されたサーバをどのように割り当てるか？
3. クライアントとサーバを結合するのはいつが適切か？
4. クライアントが実行中に結合を変更することは可能か？
5. 同じサービスを提供する複数のサーバとクライアントが同時に結合することは可能か？

これらの結合に関する課題について次に説明する。

4.13.1　サーバの名前付け

クライアントが通信したいサーバを指定することに関し主な課題は、**名前付け (naming)** である。RPC について、Birrell と Nelson[1984] は、この目的のためにインタフェース名を提案している。**インタフェース名 (interface name)** は、2 つの部分、**型 (type)** と**インスタンス (instance)** からなる。型はインタフェース自身を指定し、インスタンスはそのインタフェースでサービスを提供するサーバを指定する。例えば、インタフェースの型として**ファイルサーバ (file_server)** があり、ファイルサービスを提供するサーバにも幾つかのインスタンスがある。クライアントが特定のインスタンスを要求しなければ、インタフェース名のインスタンス部分は指定しなくても良い。

インタフェースの型の部分は、通常異なる手続きのセットやパラメータが異なる手続きを管理のために版番号を含む。分散アプリケーションプログラミングではある版を出した後アプリケーションの更新は避けられない。版番号フィールドの使用は、古いものと新しいものの共存を可能にする。新しいものが出ると古いものを更新することが望まれる。しかしながら、経験的に古いものも利用させるため古いものとの互換性は必要である。

Birrell と Nelson[1984] によると、インタフェース名のセマンティクスはイクスポータとインポータ間の調整によって決まる。したがって、インタフェース名はユーザによって生成される。これらは RPC パッケージによって決まら

ない。RPC パッケージはインポータがイクスポータを決めするためにインタフェース名を使用する場合の手段を提供する。

4.13.2 サーバの位置特定

サーバのインタフェース名は、一意の識別子である。クライアントが遠隔手続き呼び出しのためにサーバのインタフェース名を指定すると、クライアントの要求メッセージが送られる前にサーバは位置特定をしなければならない。これは主に位置特定の課題であり、このために位置特定機構 (第 10 章で述べる) が使用される。次の 2 つの共通的な方法がある。

1. **ブロードキャスティング (broadcasting)**:この方法は、指定されたサーバを位置特定するためにメッセージはクライアントによって全てのノードにブロードキャストされる。要求されるサーバは複数のノードに存在するために複数のノードから応答メッセージを受信することもある。通常、最初にクライアントノードに届いた応答がクライアントプロセスに渡され、他の応答は廃棄される。

この方式は、実装が容易であり、小規模なネットワークに適している。しかしながら、ブロードキャストの処理によって多くのノードが含まれ、メッセージトラヒックが増すため、大規模なネットワークには不向きである。したがって、名前サーバに基づく 2 番目の方法が大規模のネットワークに使用される。

2. **結合エージェント (binding agent)**:結合エージェントは、基本的にクライアントに指定されたサーバの位置特定に関する情報を提供することによってクライアントとサーバを結合するために使用される名前サーバである。この方式では、結合エージェントはサーバのインタフェースとその位置の対応を示す結合テーブルを管理する。全てのサーバは、初期化時に結合エージェントに登録される。結合エージェントに登録するために、サーバは識別情報と位置特定のためのハンドルを与える。ハンドルは、システムに依存し、イーサネットアドレス、IP アドレス、X.500 アドレス、又はノード番号とポート番号を含むプロセス識別子などである。サーバは、サービスを提供する必要がなくなると結合エージェントから登録を解除できる。結合エージェントは、定期的にサーバをポーリングし、応答がなければサーバを解除する。サーバを位置特定するために、クライアントは結合エージェントに問い合わせる。サーバが結合エー

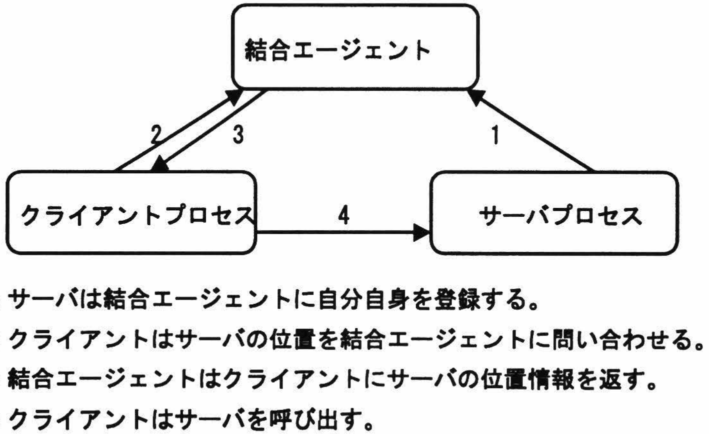


図 4.10　RPC におけるサーバ位置のための結合エージェント機構

ジェントに登録されていれば、サーバのハンドル (位置情報) をクライアントに返す。この方式を図 4.10 に示す。

結合エージェントの位置は、全てのノードが知っている。これは、結合エージェントの固定アドレスを使用するか、ノードが立ち上がった時に結合エージェントの位置を知るためのブロードキャストメッセージを使用して行われる。どちらの場合においても、結合エージェントの位置が変更されると全てのノードに新しい情報が伝えられる。

結合エージェントインタフェースは、通常 3 つのプリミティブを持つ。(a)register はサーバが結合エージェントに登録するために使用する。(b)deregister はサーバが登録を解除するために使用する。(c)lookup はクライアントがサーバの位置特定するために使用する。

サーバを位置特定するための結合エージェント機構は、幾つかの利点を持つ。1 つは、この方式は同じインタフェース型を持つ複数のサーバをサポートするため、クライアントの要求に対していずれかの利用可能なサーバが対応できる。これは、フォールトトレラント性の向上に役立つ。2 番目に、全ての結合は結合エージェントによって行われるため、複数のサーバが同じサービスを提供する場合、クライアントはサーバの負荷に応じて利用できる。3 番目は、結合エージェント機構をサーバが許容するユーザ一覧表を持つように拡張することで、承認されていないクライアントからの結合に対して結合エージェントは拒否することが可能となる。

しかしながら、結合エージェント機構には欠点もある。クライアントをサーバに結合するためのオーバヘッドが大きく、多くのクライアントプロセスが短い時間で終わる場合には問題となる。さらに、機能的な要求に加えて、結合エージェントは障害に対して強くなければならなく、性能的にボトルネックとなってはいけない。結合機能を複数の結合エージェントに分散することはこのためにも効果的である。しかし、分散によって情報の複製を持つと一貫性を保持するためにオーバヘッドが生じる。したがって、複数の結合エージェントによってもたらされる機能は、期待するほどではない。

4.13.3 結合の時期

クライアントとサーバの結合は、コンパイル時、リンク時又は呼び出し時に行われる [Goscinski 1991]。

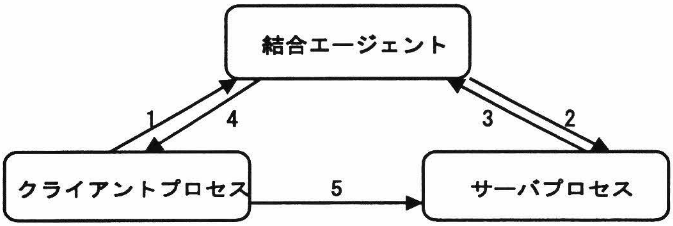
コンパイル時の結合

この方式では、クライアントとサーバモジュールは互いに関係するようにプログラムされる。例えば、サーバのネットワークアドレスはプログラマによってクライアントのコードに入るようにコンパイルでき、ファイルにサーバの名前を探すことでできる。

この方式は、サーバが移動したり、複数になったり、インタフェースが変更すると対応できなく、サーバに関するすべてのクライアントを再コンパイルしなければならない。しかしながら、この方式は限られた使用の場合には有効である。例えば、アプリケーションの構成が長期間にわたって変わらない場合である。

リンク時の結合

この方式では、サーバプロセスは初期化時に結合エージェントに登録することによってそのサービスをエクスポートする。クライアントは、呼び出しの前にサービスのために結合エージェントにインポート要求を出す。結合エージェントは、クライアントとサーバを結合し、クライアントにサーバのハンドル(サーバに呼び出しを行うために必要な詳細情報) を返す。クライアントがサーバのハンドルを受信すると呼び出しが行える。サーバのハンドルは、同じサーバに関して結合エージェントに問い合わせを行わないためにクライアントによってキャッシュに保持される。結合エージェントへの問い合わせによるオー



1：サーバは結合エージェントにサーバのインタフェース名とRPC呼び出しのパラメータを渡す。
2：結合エージェントはクライアントから受けたパラメータを含むRPC呼び出しメッセージをサーバに送信する。
3：サーバは結合エージェントに要求処理の結果を返す。
4：結合エージェントはサーバのハンドルを含む結果をクライアントに返す。
5：引き続く呼び出しはクライアントプロセスからサーバプロセスに直接送られる。

図 4.11　直接呼び出し方法による呼び出し時の結合方法

バヘッドのために、この方式は一度結合したサーバに何度も呼び出しを行う場合に適している。

呼び出し時の結合

この方式では、クライアントはサーバを呼び出す最初の時にサーバと結合する。呼び出し時の結合に共通的に使用される方法は、**間接呼び出し (indirect call)** 方式である。図 4.11 に示すように、この方式では、クライアントは最初にサーバを呼び出す時にサーバのインタフェース名と RPC 呼び出しのパラメータを結合エージェントに渡す。結合エージェントは、結合テーブルから目的のサーバの位置情報を探し、クライアントに代わって RPC 呼び出しメッセージを目的のサーバに送信する。サーバが結合エージェントに結果を返すと、結合エージェントはサーバのハンドルとともに結果をクライアントに返送し、クライアントはそのサーバへの引き続く呼び出しを直接行う。

4.13.4　結合の変更

動的に結合を変更できることは、信頼性の観点からすると非常に有効である。結合は、クライアントとサーバ間のコネクションの確立である。コネクションに関わるクライアント又はサーバは、システム状態の変化に伴って結合の変更

を要求する。例えば、複数のサーバの１つからあるサービスを受けるクライアントは、呼び出したサーバが障害となった場合に同じタイプの別のサーバに結合を変更するようにプログラムされているかもしれない。同様に、サービスが別のノードに移動したり、サーバに新しい版が実装されたような場合には、結合しているサーバが結合を変更し、クライアントが別のサーバに接続することを行うこともある。結合がサーバから変更される場合、サーバによって保持されている状態データが何も引き継がれないか、若しくは次のサーバに完全に引き継がれるかを保証することは重要である。例えば、ファイルサーバが新しいものに変更された場合、オープンされているファイルは全くない状態になるか、若しくはオープンされているファイルの全ての状態が正しく前のファイルサーバから次のファイルサーバに引き継がれなければならない。

4.13.5 複数の同時結合

システムでは、１つのサービスは複数のサーバによって提供されることもある。一般に、クライアントは同じタイプのサーバの中の１つのサーバと結合する。しかしながら、クライアントのとって同じタイプの複数のサーバと同時に結合することは有効である。論理的には、この種の結合は、呼び出しが行われるとクライアントと結合している全てのサーバは呼び出しを受信し、処理するため、マルチキャスト通信となる。例えば、クライアントが、幾つかのノードに複製されているファイルの更新を行う場合である。この場合には、クライアントはファイルの複製を持つ全てのノードのファイルサーバに同時に結合される。

4.14 例外処理

図 4.4 で、遠隔手続き呼び出しが正しく実行できない場合には、サーバは応答メッセージで誤りとなったことを通知することを説明した。また、RPC はクライアントが RPC のサーバに問い合わせできなければ障害となる。RPC システムは、このような障害をクライアントに報告するための効果的な例外処理機構を持たなければならない。このためには、起こり得る誤りのタイプに対する例外条件を定義し、誤りが発生した時にクライアントの環境内で呼び出され自動的に実行される例外処理手続きを持たなければならない。この方法では、例外処理のために言語的に構成可能なプログラミング言語を使用できる。このような言語として、Ada、CLU [Liskov et al. 1981] 及び Modula-3 [Nelson 1991,

Harbinson 1992]がある。C言語では、例外処理のために信号ハンドラが使用できる。

しかしながら、全ての言語が例外処理機構を持っているわけではない。例えば、Pascalはこのような機構を持っていない。このような言語を使用するように設計されているRPCシステムでは、一般的に例外処理に対して従来のオペレーティングシステムが提供する方法を使用する。その1つは、呼び出したプログラムに誤りタイプに応じた値を返すためのシステム呼び出しを行い、プロセスに障害を通知する。例えば、UNIXでは値－1は障害を示し、誤りタイプはグローバル変数errnoで通知される。RPCでは、誤りを示す値はサーバとの通信上の障害に関するものと、サーバから応答メッセージで通知されるものがある。誤りのタイプの詳細は、クライアントプログラムのグローバル変数に設定される値で示される。この方法は、2つの主な欠点を持っている。1つは、クライアントは返される値を絶えず調べなければならない。2つ目は、障害を示す値が実は手続きが戻す正当な値であるため十分一般的とは言えない。例えば、値として－1が障害を示す場合、この値は手続き呼び出しの結果として返される値として－5と4の和として返されることがある。

4.15　セキュリティ

RPCの実装では、呼び出しにおける暗号による**セキュリティ (security)** と同様にクライアントとサーバの**認証 (authentication)** 機能を提供するものもある。例えば、[Birrell and Nelson 1984]では、呼び出し側は呼ばれる側の識別をGrapevineの認証システム[Birrell et al. 1982]を使用することによって保証され、その逆も同様である。呼び出しや結果のエンドツーエンド暗号化に対して、米連邦政府のデータ暗号化標準[DES 1977]が[Birrell and Nelson 1984]では使用されている。暗号技術は、盗聴、改ざん、成りすましなどを防ぐ。

セキュリティの機能を持たないRPCでは、RPCのパラメータや結果は呼び出し側と呼ばれる側の間の通信をモニタすることで盗聴することができる。したがって、この場合、セキュリティを必要とするならば、認証とデータ暗号機構を実装しなければならない。アプリケーションを設計する場合、メッセージの通信に関して次のセキュリティの課題について考えることが必要である。

■クライアントによるサーバの認証は必要か？

■結果を返す時にサーバによるクライアントの認証は必要か？

■呼び出し側や呼ばれる側以外のユーザに RPC のパラメータや結果が渡されることは正しいか？

これらを含むセキュリティの課題について第 11 章で詳細に説明する。

4.16 特殊な RPC

4.16.1 コールバック RPC

通常の RPC プロトコルでは、呼び出し側と呼ばれる側のプロセスはクライアントとサーバの関係を持つ。これとは違って、**コールバック RPC (callback RPC)** は関係するプロセス内でピアツーピアの機能を提供する。このため、この機能では 1 つのプロセスがクライアントとサーバとになれる。

コールバック RPC 機能は、分散アプリケーションでは非常に有効である。例えば、時系列的にユーザの入力を必要とする場合や特殊な状態で更なる処理を行う場合、遠隔で相互に処理を行うアプリケーションは、このようなタイプの機能を必要とする。図 4.12 に示すように、このようなアプリケーションでは、クライアントプロセスがサーバプロセスに RPC を行い、そのサーバプロセスがこの要求を実行するためにクライアントプロセスに RPC を行う。クライアントプロセスは、サーバプロセスの要求に基づいて必要な処理行い、サーバプロセスにコールバック RPC の応答を返す。この応答を受けると、サーバプロセスは手続きの実行を再開し、最初の呼び出しに対する結果をクライアントに返送する。サーバはクライアントに最終の結果を返す前に、クライアントに複数のコールバックを行うこともある。

サーバがクライアントにコールバックする機能は重要であり、RPC プロトコルの設計においてこの機能を可能にするように考慮することが必要である。特に、コールバック RPC 機能を提供するためには、次のことが必要になる。

■サーバへのクライアントのハンドルの提供
■クライアントプロセスのコールバック RPC の待機
■コールバックのデッドロックの処理

これらの課題に対する処理方式について以下に説明する。

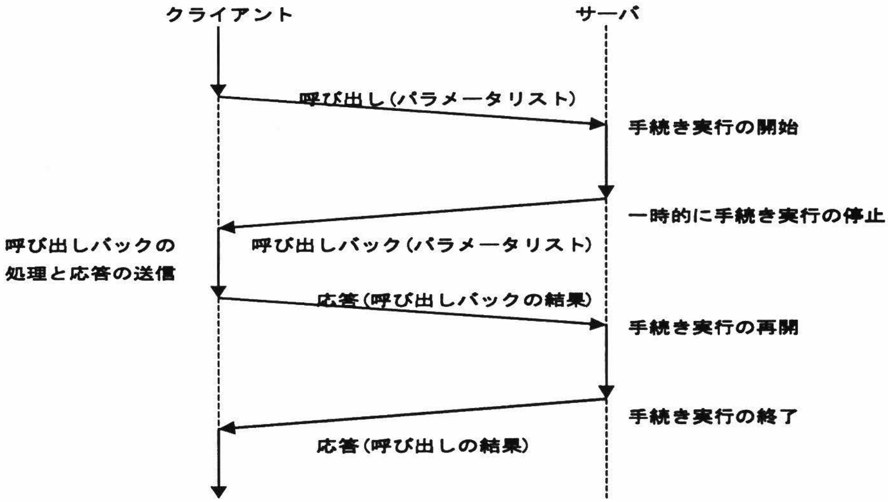


図 **4.12**　呼び出しバック RPC

サーバへのクライアントのハンドルの提供

サーバは、クライアントにコールバックを行うためにクライアントのハンドルを持たなければならない。クライアントのハンドルは、ユニークにクライアントプロセスを識別し、サーバに呼び出すための十分な情報を提供する。典型的には、クライアントプロセスは、コールバックサービスのために一時的なプログラム番号を使用し、結合エージェントにこのプログラム番号を登録することによってコールバックサービスをエクスポートする。プログラム番号は、サーバに RPC 要求の一部として送られる。コールバック RPC を行うために、サーバは与えられたプログラム番号を用いてクライアントに通常な RPC 要求を行う。クライアントがサーバにプログラム番号を送る代わりに、そのハンドル、例えばポート番号を送ることもできる。クライアントのハンドルは、サーバが直接クライアントと通信することによって使用することができ、結合エージェントを用いないでも実現できる。

クライアントプロセスのコールバック **RPC** の待機

クライアントプロセスは、サーバからの RPC 要求を処理し、サーバからのコールバック RPC に対してクライアントプロセスによって行われた処理としてサーバに応答を返すためにもコールバックを待たなければならない。コール

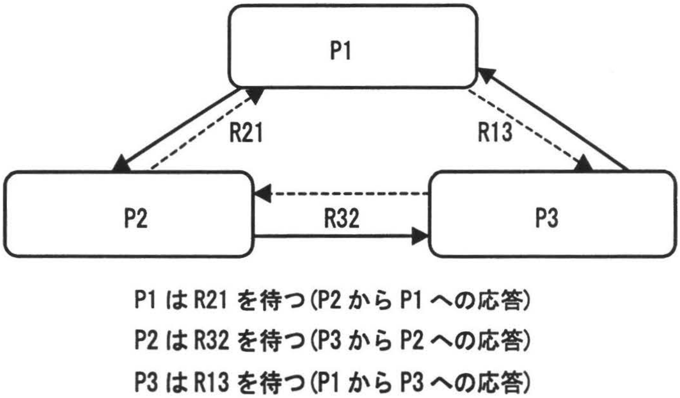


図 4.13 コールバック RPC 機構におけるコールバックのデッドロックの例

バックを待つために、クライアントプロセスは svc ルーチンを呼び出す。この svc ルーチンは要求を受信するまで待ち、受信すると適切な手続きに要求を渡す。

コールバックのデッドロックの処理

コールバック RPC において、プロセスはクライアントとサーバの役割を演じるため、コールバックのデッドロックが起きる。例えば、プロセス P_1 がプロセス P_2 への RPC 呼び出しを行い、P_2 からの応答を待つような最も単純な場合を考える。このときに、プロセス P_2 がプロセス P_3 に RPC 呼び出しを行い、P_3 からの応答を待つ。またこのときに、プロセス P_3 がプロセス P_1 に RPC 呼び出しを行い、P_1 からの応答を待つ。しかし、P_1 は P_2 への要求が満たされなければ P_3 の要求を処理できないし、P_2 も P_3 への要求が終わらなければ P_1 の要求を処理できなく、P_3 も P_1 への要求が終わらなければ P_2 の要求を処理しない。図 4.13 に示すように、P_1 は P_2 からの応答を待ち、P_2 は P_3 からの応答を待ち、P_3 は P_1 からの応答を待つような状況になってしまう。この結果、3 つのプロセスとも要求が満たされなく、永久に待ちつづけることになる。実際、コールバックのデッドロックは 3 つのプロセスの相互の従属関係によって発生する。

コールバック RPC を使用する上でデッドロックが起きないように処理しなければならない。このための処理についての各種の方式を第 6 章に述べる。

4.16.2　ブロードキャスト RPC

RPC の IPC は、一般に 1 つのクライアントプロセスと 1 つのサーバプロセスからなる 1 対 1 のタイプである。しかしながら、既に説明したように性能の観点からより高度な並列処理を行う分散アプリケーションは、通信システムがブロードキャストやマルチキャスト通信機の追うを提供することを要求する。RPC に基づく IPC 機構は、このようなアプリケーションに対して通常ブロードキャスト RPC 機能を提供する。**ブロードキャスト RPC (broadcast RPC)** では、クライアントの要求はブロードキャストで送信され、その要求を処理するために関連する手続きを持つ全てのサーバによって処理される。クライアントは多くからの応答の受信を待つ。

ブロードキャスト RPC 機構は、クライアントの要求をブロードキャストするために次の 2 つの方式のいずれかを使用する。

1. クライアントは、要求メッセージがブロードキャストされることを示すために特別なブロードキャストプリミティブを使用する。要求は結合エージェントに送られ、登録されている全てのサーバに要求は転送される。この方式では、全てのブロードキャスト RPC メッセージは結合エージェントに送られるため、結合エージェントに登録されているサービスのみがこのブロードキャスト RPC で使用できる。

2. 次の方式は、ブロードキャストポートを宣言するものである。各ノードのネットワークポートはブロードキャストポートに接続される。ノードのネットワークポートは、ブロードキャストメッセージをキューイングする場所となる。ブロードキャスト RPC を行うクライアントは、まずブロードキャストポートとの結合を行い、このポートに RPC メッセージを送信することでブロードキャストを行う。単一 RPC とブロードキャスト RPC に対して同じプリミティブが使用できる。さらに、1 番目の方式とは異なり RPC メッセージが対応できるサーバの範囲にのみ送信するようなマルチキャスト RPC に拡張できる。このために、ポート宣言機構は新たなマルチキャストポートに対して限定したサーバを関係付けることができる柔軟性を持たせることが必要である。

ブロードキャスト RPC メッセージはネットワークの全てのノードに送信されるため、各ノードから応答が返る。既に述べたように、要求される信頼性の

程度によってクライアントプロセスは0、1、n 中 m、又は全てを待つことができる。ある実装では、ブロードキャスト RPC をサポートするサーバは、要求が正しく処理できた時に応答を返し、そうでない場合には何も応答を返さない。このようなシステムは、ブロードキャスト RPC プロトコルの信頼性を改善するためにタイムアウトによる再送機構を使用する。例えば、SunOS では、ブロードキャスト RPC プロトコルはブロードキャストを行い、要求を再送する前に 4 秒待つ。次にタイムアウト時間が 14 秒になるまでタイムアウトになるたびに 2 秒加算する。したがって、最悪の場合、要求は 6 回送信され、全体で 54 秒 (4+6+8+10+12+14) 待つことになる。SunOS では、ブロードキャスト RPC は信頼性の低いパケット転送のプロトコルを使用し、再送を行う。再送間隔が増加する方式は、**バックオフアルゴリズム (back-off algorithm)** と呼ばれる。タイムアウトによる再送でのバックオフアルゴリズムの使用は、物理ネットワークのトラヒック負荷を抑えることができる。

4.16.3 バッチモード RPC

バッチモード RPC (batch-mode RPC) は、クライアントの転送バッファに別の RPC 要求をキューイングするために使用され、それをまとめてサーバに送信する。これは、次の 2 つの目的を持っている。

1. 各 RPC 要求を個別にサーバに送信し、各応答を受信することによるオーバヘッドを軽減する。
2. 呼び出し率が高い (毎秒 50 から 100 の遠隔呼び出し) アプリケーションを RPC として実現することは困難である。このようなアプリケーションは、バッチモード RPC が向いている。

しかしながら、バッチモード RPC はサーバに多くの要求を行い、その応答の順序を気にしないクライアントのアプリケーションに限られる。したがって、要求はクライアント側でキューイングされ、次のような条件になった時にサーバにまとめて送信される。

1. あらかじめ決められた時間が経過した。
2. あらかじめ決められた数の要求がキューイングされた。
3. バッファ容量が一杯になった。

4. 呼び出しは、結果が期待できるサーバの手続きに対して行われる。プログラミングの立場からすると、このような呼び出し (キューイングされない RPC 要求) のセマンティクスは、サーバはキューイングされる要求とは区別し、クライアントに応答を返す。

上記の 1、2、又は 3 によってキューイングされている要求の送信は、キューイングされない要求とは独立に行われ、クライアントには通知されない。

キューイングされたメッセージは信頼性のある転送が必要である。したがって、バッチモード RPC 機構は TCP のような信頼性のあるトランスポートを必要とする。さらに、バッチモードとしてシンタックスの透過性は必要とするが、他のクライアントが同時にサーバにアクセスする場合、処理のタイミングが不明確になる。

4.17　不均質な環境における RPC

アプリケーションの移植性を高めるためには、分散アプリケーションの設計において**不均質性 (heterogeneity)** は重要な課題である。不均質な環境で RPC システムを設計する場合、不均質性について考慮すべき 3 つの共通的な事柄を次に述べる。

1. **データ表現 (data representation)**:異なるアーキテクチャを持つマシンは、異なるデータ表現を使用する。例えば、整数の表現としてあるマシンでは下位バイトアドレスに最下位桁が、別のマシンでは上位バイトアドレスに最下位桁が置かれる。同様に、整数はあるマシンでは 1 の補数で、別のマシンでは 2 の補数で表現されることもある。浮動小数点の表現もマシンのアーキテクチャが変われば異なる。したがって、不均質な環境における RPC システムは手続き呼び出しにおいてクライアントとサーバのデータ表現に注意して設計しなければならない。

2. **トランスポートプロトコル (transport protocol)**:アプリケーションの移植性のためには、RPC システムは下位のネットワークトランスポートプロトコルに依存すべきではない。これによって、RPC システムを使用するアプリケーションが別のトランスポートプロトコルを使用するネットワークを利用できる。

3. 制御プロトコル (**control protocol**):アプリケーションの移植性のために、RPC システムは各トランスポートパケット内の制御情報を定義するネットワーク制御プロトコルにも依存すべきでない。

不均質な環境の RPC システムを設計する上でこのような考え方がとられるが、実際結合されるまでデータ表現、トランスポートプロトコル、制御プロトコルの選定は決まらない。従来の RPC システムでは、これらの決定は RPC システムを設計する時点で行われた。すなわち、不均質な環境における RPC システムの結合機構は従来の RPC システムで使用される結合機構よりも情報的に複雑になっていると考えられる。これは、どのデータ変換ソフトウェア (変換が必要ならば)、どのトランスポートプロトコル、どの制御プロトコルをクライアントとサーバ間で使用し、結合呼び出しの結果パラメータとしてスタッブに正しい手続きを返すべきかを決定するための機構を含んでいる。このような結合機構の詳細は、ユーザには透過である。すなわち、アプリケーションプログラムは結合機構の構成要素の構造に直接アクセスすることはない。アプリケーションは、基本的な動作として結合し、RPC システムの呼び出しを介して問い合わせや拒絶を行う。

不均質な環境に対して設計された RPC システムとして、HCS (Heterogeneous Computer System) RPC (HRPC と呼ばれる) [Bershad et al. 1987]、DEC SRC (System Research center) Firefly RPC [Schroeder and Burrows 1990]、Matchmaker [Jones et al. 1985]、Horus [Gibbons 1987] がある。

4.18 ライトウエイト RPC

ライトウエイト RPC (lightweight RPC)(LRPC) は、Bershad 等 [1990] によって紹介され、DEC SRC Firefly マイクロプロセッサのワークステーションの Taos オペレーティングシステムに導入された。以下に示す内容は、それに関する論文 [Bershad et al. 1990] に基づいている。

第 1 章で述べたように、カーネルの規模によってオペレーティングシステムは大きく 2 つ、モノリシックカーネルのオペレーティングシステムとマイクロカーネルのオペレーティングシステムに分類される。モノリシックカーネルのオペレーティングシステムは、大きく一体型のカーネルであり、単純なハードウェア上に実現されている。一方、マイクロカーネルのオペレーティングシステムでは、小さなカーネルが基本的な機能を提供し、サービスのほとんどは

ユーザレベルのサーバとして提供される。これらのサーバは、一般にプロセスとして実装され、別々にプログラムすることができる。各サーバはオペレーティングシステムの構成要素を形成し、それ自身のアドレス空間を持つ。モノリシックカーネルに比べて、この方式ではオペレーティングシステムの各構成要素が相互に通信するためにIPCの形態を使用するためにサービスとして提供されるものは少ない。この方式の利点は、単純で柔軟性に富んでいる点である。モジュール構造のためマイクロカーネルのオペレーティングシステムは、設計、実装及び保守管理が単純で容易である。

マイクロカーネルの場合、オペレーティングシステムの各構成要素がそれ自体のアドレス空間を持つならば、各アドレス空間はドメインを構成し、メッセージは全てドメイン **(domain)** 間通信によって行われる。この場合、オペレーティングシステムの通信トラヒックは次の 2 つのタイプになる [Bershad et al. 1990]。

1. **ドメイン間 (cross-domain)**、同一マシン内の異なるドメイン間での通信
2. **マシン間 (cross-machine)**、別のマシンのドメイン間での通信

LRPCは、ドメイン間の通信に設計され、最適化されている通信機能である。

従来の RPC システムは、ドメイン間通信及びマシン間通信に使用できるが、マシン間通信ができる従来の RPC システムをドメイン間通信に適用した場合、不必要にコストが高くなることを Bershad 等は指摘している。このために、システム設計者は安全性や性能を犠牲にしてマイクロカーネルのオペレーティングシステムの関連する構成要素を 1 つのドメインに押し込めてしまう。したがって、マイクロカーネルの方式を使用する基本的な利点は、十分に生かされない。このため、Bershad 等は従来の RPC システムよりも良い性能を持つドメイン間通信のための LRPC 機能を設計した。さらに、LRPC は、安全性に富み、透過性を持ち、またマイクロカーネルのオペレーティングシステムにおける有望な通信機能である。

従来の RPC システムの性能を向上させるために、LRPC では次の 4 つの記述が使用される。

4.18.1 単純制御転送

LRPCは、可能な限り、従来のRPCシステムよりも単純な制御転送機構を使用する。例えば、ハンドオフスケジューリングと呼ばれる特別なスレッドスケジューリング機構(スレッド及びハンドオフスケジューリング機構の詳細については第8章で説明する)を使用し、LRPCにおけるクライアントスレッドからサーバスレッドへの直接的なコンテキスト切り替えを行う。この機構では、クライアントがサーバの手続きを呼び出した時、サーバにパラメータのスタックと実行スレッドを提供する。呼び出しによってカーネルにトラップが発生する。カーネルは呼び出される側を調べ、呼び出しの関係を生成し、クライアントのスレッドを直接サーバのドメインに渡す。これにより、サーバは即座に実行を開始する。呼び出された手続きが完了すると、制御と結果がカーネルを通してクライアントの呼び出した点に戻される。これに比べて、従来のRPCでは、RPCのクライアントとサーバのスレッド間のコンテキスト切り替えは、クライアントのスレッドとサーバのスレッドがそれぞれのドメインに固定であり、ある別の信号によってスケジューラはクライアントをブロックするためにシステムデータ構造を操作し、実行のためにサーバのスレッドを選択するため、時間がかかっていた。

4.18.2 単純データ転送

RPCにおいて、パラメータと結果はメッセージの形式でクライアントとサーバのドメイン間を渡す必要がある。従来のRPCシステムに比べて、LRPCはドメイン間のデータのコピー回数を少なくすることでデータ転送のコストを軽減する。例えば、従来のドメイン間のRPCでの手続きのパラメータの受け渡しを考える。図4.14(a)に示すように、この場合のパラメータは、4回のコピーが行われる。

1. クライアントのスタックからRPCメッセージ
2. クライアントドメインのメッセージからカーネルドメインのメッセージ
3. カーネルドメインのメッセージからサーバドメインのメッセージ
4. サーバドメインのメッセージからサーバのスタック

このデータ転送動作を単純にするために、LRPCはクライアントとサーバが

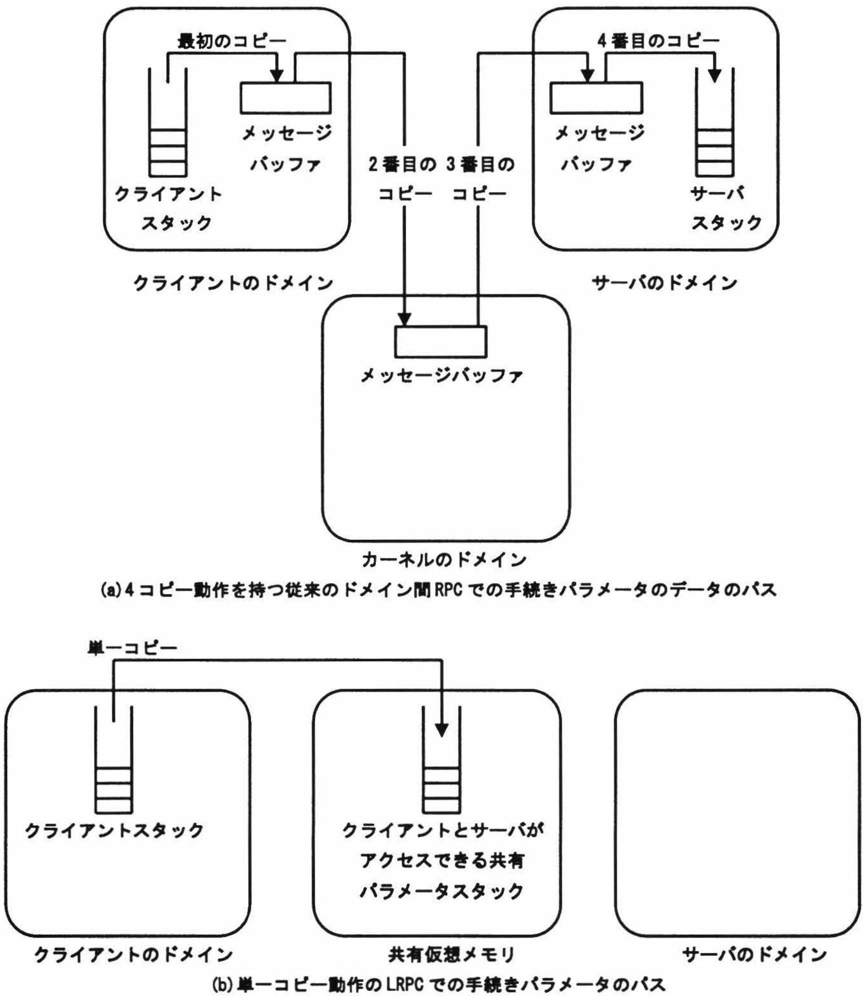


図 4.14　従来のドメイン間 RPC と LRPC におけるデータ転送機構

アクセスできる共有パラメータスタックを使用する。したがって、図 4.14(b) に示すように、LRPC の同じパラメータはコピーがクライアントのスタックから共有パラメータスタックへの 1 度になる。サーバは、パラメータスタックからパラメータを使用する。パラメータスタックのこのような配置により、LRPC はクライアントとサーバ間に個別のチャネルを提供でき、正確性や安全な動作を保証するために必要となるパラメータや結果のコピーを可能にする。

4.18.3 単純スタッブ

ドメイン間とマシン間の呼び出しの相違は、通常、RPC システムの下位のレベルによるスタッブに対する透過性の違いである。これにより、汎用的なインタフェースと実行が行われるが、まれにしか要求されない。

LRPC での制御とデータの転送の単純なモデルの使用は、より最適化された汎用的なスタッブを実現する。全ての手続きは、クライアントのドメインに呼び出しスタッブとサーバのドメインにエントリスタッブを持つ。LRPC インタフェースでは、各手続きに対して 3 階層の通信プロトコルが定義される。

1. エンド対エンド、プログラミング言語やアーキテクチャの従来の呼び出し方法によって記述される。
2. スタッブ対スタッブ、スタッブ自身で実装される。
3. ドメイン対ドメイン、カーネルによって実装される。

階層間のやり取りのコストを軽減するために、LRPC スタッブはプロトコル層間の境界を明確にしない。例えば、制御を転送する場合、カーネルは呼ばれるサーバの手続きに渡すべき最初の呼び出しフレームを実行スタックに渡し、サーバのドメイン内の対応する手続きのエントリを呼び出す。メッセージの検査などは行われないで、サーバスタッブは手続きの最初の命令に直接飛び、手続きの実行を開始する。この方法で、単純な LRPC は、1 つの正式な手続き呼び出し (クライアントスタッブへの) と 2 つの返し (サーバ手続きからとクライアントスタッブから) のみを必要とする。

4.18.4　並行性の設計

LRPC のクライアントとサーバのプロセスのノードが共有メモリを持つ複数のプロセッサからなる場合、単一プロセッサノードよりも高い処理性能と小さい遅延を実現するために特別な機構が使用される。処理性能は、重要となるドメイン間の転送で共有データ構造の使用を少なくし、不必要なロック競合を避けることによって向上することができる。一方、空きのプロセッサでのドメインのキャッシングにより、コンテキスト切り替えのオーバヘッドを軽減することで遅延を小さくすることができる。これは、起動時の遅延を少なくするために空きのプロセッサでのブロックされているスレッドをキャッシングすることでオペレーティングシステムの処理遅延を軽減する一般的な考え方に基本的基づいている。スレッドに代えて、LRPC はドメインをキャッシュし、空きのドメインのコンテキスト内で実行するために必要なスレッドが即座に行い、このスレッドは最後に実行していたものではない。

Bershad 等 [1990] によって行われた性能評価に基づくと、LRPC は従来の方式で 3 つの性能向上を達成している。このように、LRPC は、ドメイン間通信のコストを従来のハードウェアによって実現できるレベルに軽減している。

4.19　性能のための最適化

ソフトウェアの設計と同じように、性能は分散アプリケーションの設計において課題である。LRPC は、RPC を使用する分散アプリケーションの性能を向上されるための幾つかの最適化を行っている。RPC に基づく分散アプリケーションを設計する上で性能に関わる他の最適化について以下に説明する。

4.19.1　複数サーバへの並列アクセス

RPC の利点の 1 つは同期制御機能にあるが、多くの分散アプリケーションは複数のサーバに並列アクセスを行うことができる。この機能を提供するために、次の 3 つの方法の 1 つが使用される。

1. 異なるサーバに遠隔手続き呼び出しを独立して行う各スレッドにおいてクライアントプロセスの実装でこのようなスレッド (第 8 章で説明) を使用する。この方法は、下位のプロトコルのアドレス付けが応答を正しく中継することを

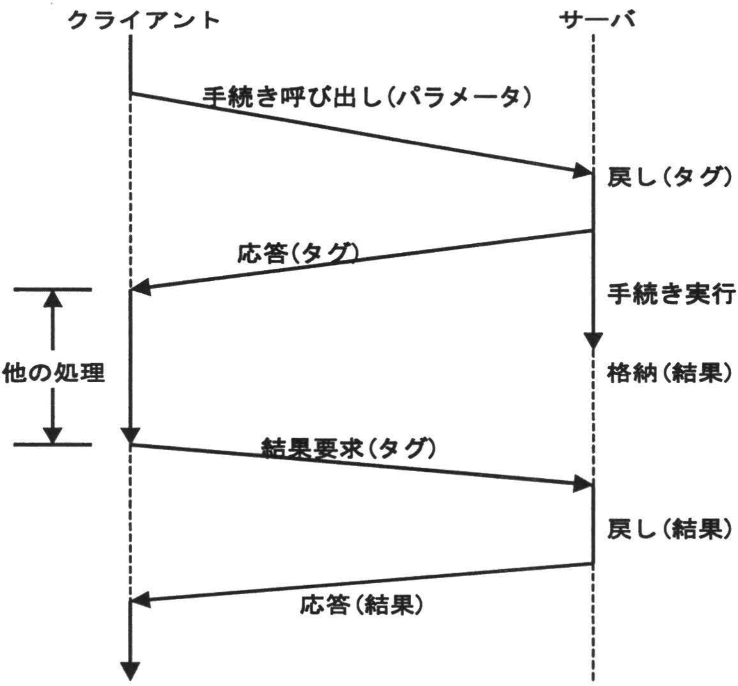


図 **4.15** 複数サーバへの並行サクセス機能を提供する早期応答方式

要求する。

2. 次の方法として、早期応答方法がある [Wilbur and Bacarisse 1987]。図4.15 に示すように、この方法では、呼び出しは 2 つの RPC 呼び出しに分けられ、1 つはサーバにパラメータを渡すために、もう 1 つは結果を要求するために使用される。最初の呼び出しに対する応答では、サーバは正しい結果と対応させるために 2 番目の呼び出し時に返されるタグを返す。クライアントは、2 つの呼び出し間の時間的な遅延を決定し、この時間の間に別のこと、恐らく別の RPC 呼び出しを行う。この方法の欠点は、サーバがクライアントが要求するまで結果を保持しなければならないことである。したがって、結果の要求が遅れると、サーバ側で輻輳や不必要なオーバヘッドを引き起こす。

3. 3 番目の方法は、呼び出しバッファリングと呼ばれるもので、Gimson[1985]

によって提案された。この方法では、クライアントとサーバは直接相互作用を持たない。これらは、呼び出しバッファサーバを介して間接的に交信する。RPC 呼び出しを行うために、クライアントは呼び出し要求を呼び出しバッファサーバに送信し、サーバやクライアントの名前と一緒にパラメータがバッファに置かれる。クライアントは、RPC 呼び出しの結果が必要になるまで別の処理を行うことができる。クライアントが結果が必要となる状態になると、呼び出した結果があるか定期的に呼び出しバッファサーバのポーリングし、結果があればそれを受け取る。サーバ側では、サーバの処理がなくなると定期的に呼び出しバッファサーバを要求があるかを調べるためにポーリングする。もし要求があれば、要求を受け、実行し、実行結果を呼び出しバッファサーバに返すために呼び出しバッファサーバをコールバックする。この方法を図 4.16 に示す。

この方法の応用として、非同期 RPC をサポートするために MIT で開発された Mercury 通信システム [Liskov and Shrira 1988] がある。この Mercury 通信システムは、promise と呼ばれる新しいデータ型を持ち、この型は RPC 呼び出し中に生成され、遠隔手続きの結果や例外に対応する型を与えられる。結果が到着すると、適切な promise のデータに格納され、呼び出し側が結果を読み出す。したがって、呼び出しを行うと、呼び出し側は別の処理を行い、呼び出し結果を適切な promise から読み出すことがきる。

promise は、2 つの状態、ブロック又はレディのどちらかをとる。生成されてから呼び出し結果を到着するまではブロック状態にあり、その後がレディ状態になる。レディ状態の promise は変化しない。

呼び出し側が promise の状態を調べるためや呼び出し結果を要求するために 2 つの動作 (ready と claim) が提供されている。ready 動作は、promise の状態 (ブロック／レディ) を調べるために使用される。レディならば真、ブロックならば偽の値が返される。claim 動作は、呼び出し結果を promise から得るために使用される。claim 動作は、promise が呼び出し結果を返せる状態のレディになるまで呼び出し側をブロック状態にする。したがって、呼び出し側は promise がレディになるまで別の処理を行い、定期的に ready 動作で promise の状態を調べることができる。

4.19.2 複数の要求の同時処理

RPC システムでは、次のような遅延が起こり得る。

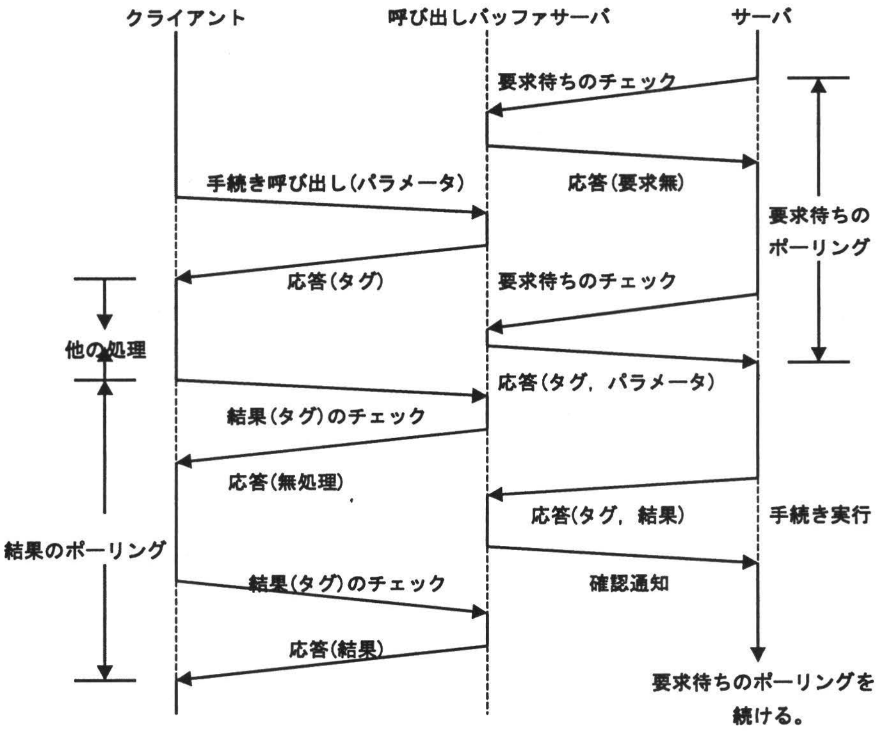


図 **4.16** 複数サーバへの並行アクセス機能を提供する呼び出しバッファ方式

1. 資源が一時的に使用できないためにサーバが待つことのよる遅延。例えば、呼び出し実行中に、サーバはロックされている共有ファイルをアクセスする場合には待たなければならない。

2. サーバがある程度の処理時間や転送時間が必要となる遠隔機能を呼び出した時に発生する遅延。

性能を向上させるために、RPC の実装として他の要求の処理の完了を待つのではなく、サーバは複数の要求を受理し、処理できる機構を持つ必要がある。これは、サーバが複数の要求を同時にサービスできるようにサーバを設計することを要求する。これを達成するためには、サーバの実装において動的なスレッド生成機能を持つマルチスレッドサーバの方法がある (この方法の詳細については第 8 章で説明する)。

4.19.3　サーバにおける呼び出し当たりの処理負荷の軽減

多くのクライアント要求が発生すると、サーバが各要求に対して多くの処理を行う場合サーバの性能に深く影響を与える。そのため、RPC システムの全体の性能を改善するためには要求を保持する時間を短くし、サーバが処理する量を少なくすることが重要である。これを実現する方法の 1 つに、ステートレスサーバの使用があり、クライアントがサーバに送信する要求の状態を管理する方法である。この方法は、多くの場合、アプリケーションのクライアントの部分はクライアントとサーバ間の情報の流れを制御するものであり、受け入れ易い。

4.19.4　繰り返し等価な遠隔手続きの応答のキャッシング

繰り返し等価な遠隔手続きにおいて、ちょうど 1 回セマンティクスを達成するために応答をキャッシュする方法は、既に説明している。しかしながら、応答のキャッシュはサーバが過負荷状態にある場合にサーバの性能を改善する上で、繰り返し等価な遠隔手続きと関連する。サーバが要求を処理できる以上にクライアントが要求を出すと、クライアントの処理のタイミングが狂い、要求の再送となり、ますます状況を悪化させる。このような状況では、応答のキャッシュによりサーバでの要求の処理を 1 回のみにすることができる。クライアントが要求を再送したとしても、サーバはキャッシュの応答を返すだけで良い。

4.19.5 タイムアウト値の選択

障害処理のために、タイムアウトによる再送は、分散アプリケーションでは必要である。ここでの問題は、タイムアウト値をどのように選ぶかである。小さ過ぎるタイムアウト値は、タイムアウトを頻繁に発生させ、不必要な再送を引き起こす。一方、大きすぎるタイムアウト値は、メッセージが実際に紛失した場合に不必要な長い遅延を発生させる。RPC システムでは、サーバは個々の要求をサービスする上でいろいろ処理時間は変わり、サーバの負荷、ネットワーク中継、ネットワークの輻輳にも依存する。クライアントが応答が受け取れないために要求を何度も再送すると、サーバの負荷やネットワークのトラヒックはますます悪い状態になる。このような状況を防ぐために、タイプアウト値を適切に選択することが重要となる。この問題を解決するための方法として、タイムアウト値を指数関数的に増加させるバックオフ方式が有効である。

4.19.6 RPC プロトコル仕様設計

性能を高めるために、RPC システムのプロトコル仕様は、ネットワークに転送されるデータ量を最少にするために設計されなければならない。転送されるデータ量を軽減することは、次の 2 つの点で有効になる。データの符号化及び復号の処理時間の短縮とネットワークでのデータ転送時間の短縮である。幾つかの既存の RPC システムは、既存の UNIX システムやインターネットのネットワークに適合するために基本的なプロトコルとして TCP/IP 又は UDP/IP を使用する。これは、クライアントやサーバを UNIX システムや標準のネットワークで動作するようにすることを意味している。しかしながら、RPC に対して汎用的な標準プロトコルを使用することは、各種の機能を持ち合わせている汎用プロトコルであるために性能の低下を招きかねない。例えば、IP スウィート (TCP/IP 又は UDP/IP に属する) のパケットは全体で 13 のヘッダフィールドを持つが、そのうちの 3 フィールドが RPC に関するものであり、送信元アドレス、宛先アドレス及びパケット長である。しかしながら、これらのヘッダフィールドのあるもの、例えば分割や検査合計に関するものは、正しい IP パケットを処理するために送信側で設定され、受信側で精査させる。これらのフィールドのあるもの、例えば検査合計は、処理時間がかかる。したがって、性能を高めるためには、RPC システムは特別な RPC プロトコルを使用すべきである。もちろん、このために新しいプロトコルは基本から設計され、実装、

テスト、既存システムへの適用を行い、よりよい処理を実現することが必要である。

4.20　事例：SUN RPC、DCE RPC

多くの RPC システムが、構築され、今日使用されている。このようなシステムとして、Cedar RPC システム [Birrell and Nelson 1984]、Xerox NS ファミリプロトコルの Courier[Xerox Corporation 1981]、Eden システム [Almes et al. 1985]、CMU Spice システム [Jones et al. 1985]、Sun RPC[Sun Microsystems 1985]、Argus[Liskov and Scheifler 1983]、Arjuna[Shrivastava et al. 1991]、HP 研究所で開発された研究システム [Gibbons 1987]、NobelNet ' sEZ RPC[Smith 1994]、Apollo のネットワークコンピューティングアーキテクチャ(NCA) から発展した Open Software Foundations(OSF) の DCE RPC[Rosenberry et al. 1992]、それにワシントン大学で開発された HRPC システム [Bershad et al. 1987] がある。この中で、最も良く知られている UNIX RPC システムが Sun RPC である。したがって、Sun RPC は本章で事例として説明する。さらに、DCE 構成要素を説明するために DCE RPC についても簡単に述べる。

4.20.1　Sun RPC

スタッブ生成

Sun RPC は、ユーザ自身がスタッブを書けば柔軟性があるが、自動的なスタッブ生成法をとる。アプリケーションのインタフェース定義は、RPC 言語 (RPCL) と呼ばれる IDL によって書かれ、この RPCL は、外部データ表現を記述するために設計された Sun XDR 言語を拡張したものである。例えば、4.8.1 節で述べたステートレスのファイルサーバのインタフェース定義は、図 4.17 のようになる。この図に示すように、インタフェース定義は、サービスのプログラム番号 (ここの例では 0x20000000) と版番号 (ここでは 1)、サービスがサポートする手続き (この例では READ と WRITE)、各手続きに対する入力及び出力パラメータ、及びサポートする型定義を含む。3 つの値、プログラム番号 (STATELESS_FS_PROG)、版番号 (STATELESS_FS_VERS) 及び手続き番号 (READ 又は WRITE) によってユニークに遠隔手続きを識別する。READ と WRITE 手続きにはそれぞれ番号 1、2 が与えられる。番号 0 は、ヌル手続き用に取られており、これは自動的に生成され、サーバが動作可能かを試験

するために使用される。インタフェース定義ファイル名は拡張子 x(例えば、StatelessFS.x) を持つ。

```
/*  ファイル StatelessFS.x におけるステートレスのファイルサービス
(StatelessFS)のためのインターフェース定義  */
const FILE_NAME_SIZE = 16
const BUFFER_SIZE = 1024

typedef string FileName<FILE_NAME_SIZE>;
typedef long Position;
typedef long Nbyte;

struct Data{
          long n;
          char buffer[BUFFER_SIZE];
};

struct readargs{
            FileName          filename;
            Position           position;
            Nbytes             n;
};

struct writeargs{
            FileName          filename;
            Position           position;
            Data              data;
};

program STATELESS_FS_PROG{
          version STATELESS_FS_VERS{
                              Data           Read(readargs) = 1;
                              Nbytes         WRITE(writeargs) = 2;
} = 0x200000000;
```

図 4.17 Sun RPC の LRPC で書かれたステートレスのファイルサービスのインタフェース定義

Sun RPC の IDL コンパイラは、rpcgen と呼ばれる。インタフェース定義ファイルから rpcgen は次のものを生成する。

1. インタフェース定義ファイルで定義された共通の定数と型の定義を含む

ヘッダファイル。これは、自動的に生成される全ての XDR の結集及び非結集の手続きに対する外部宣言も含む。ヘッダファイルの名前は、rpcgen への入力ファイル名を基本として拡張子。h が付与される (例えば、statelessFS.h)。このファイルは、クライアントとサーバのプログラムファイルに手動で入れられ、# include を使用するクライアントスタッブ、サーバスタッブ及び XDR フィルタファイルに自動的に入る。

2. XDR の結集及び非結集手続きを含む XDR フィルタファイル。これらの手続きは、クライアントとサーバスタッブ手続きによって使用される。このファイル名は、rpcgen への入力ファイル名を基本として拡張子_xdr.c が付与される (例えば、StatelessFS_xdr.c)。

3. インタフェース定義ファイルで定義される各手続きのスタッブ手続きを含むクライアントスタッブファイル。クライアントスタッブ手続き名は、インタフェース定義で与えられる手続きの名であり、小文字に変換され、版番号が付けられる。すなわち、この例では、READ と WRITE 手続きのクライアントスタッブ手続き名は、それぞれ read_1 と write_1 になる。クライアントスタッブファイル名は、rpcgen への入力ファイル名を基本として拡張子_clnt.c が付与される (例えば、tatelessFS_clnt.c)。

4. main ルーチン、dispatch ルーチン、インタフェース定義ファイルで定義される各手続きのスタッブ手続き及びヌル手続きを含むサーバスタッブファイル。main ルーチンはトランスポートハンドルを生成し、サービスを登録する。既定値として UDP と TCP トランスポート上のプログラムが登録される。しかしながら、ユーザは rpcgen のコマンドラインオプションによってどちらかのトランスポートを選択することができる。

dispatch ルーチンは、遠隔手続き呼び出しを適切な手続きに引き渡す。このルーチンで使用される名前は、プログラム名を小文字に変換し、版番号を付けて表させる (例えば、stateless_fs_prog_1)。

サーバスタッブファイル名は、rpcgen へのファイル名を基本として拡張子_svc.c が付与される (例えば、StatelessFS_svc.c)。

rpcgen によって生成されるファイルを使用して、RPC アプリケーションは次の方法で生成される。

```
/* ファイルclient.cにおけるステートレスのファイルサービスに対するクライアントソースプログラムの概観 */
#include <stdio.h>
#include <rpc/rpc.h>
#include "StatelessFS.h"
main (argc, argv)
        int argc;
        char **argv;
{
        CLIENT          *client_handle;
        char            *server_host_name = "paris";
        readargs        read_args;
        writeargs       write_args;
        Data            *read_result;
        Nbytes          *write_result;

        Client_handle = clnt_create (server_host_name, STATELESS_FS_PROG,
                        STATELESS_FS_VERS, "udp");
                /* クライアントハンドルを得る。ソケットを生成する。*/
        if (client_handle == NULL){
                clnt_pcreateerror (server_host_name);
                return(1);  /*サーバをコンタクトできない*/
        };

        /* パラメータを準備し、read手続きRPCを作る */
        read_args.filename = "example";
        read_args.position = 0;
        read_args.n = 500;
        read_result = read_1 (&read_args, client_handle);
        ...
        ...
        ...

        /* パラメータを準備し、write手続きRPCを作る */
        write_args.filename = "example";
        write_args.position = 501;
        write_args.data.n = 100;
        * &write_args.data.bufferのデータの100バイトを出すための文 */
        write_result = write_1 (&write_args, client_handle);
        ...
        ...
        ...
        clnt_destroy(client_handle);
        /* 終了時にクライアントハンドルを終わらせ、ソケットを閉じる */
```

0.6

図 4.18 図4.17のステートレスのファイルサービスのクライアントソースプログラムの概略

```
/* ファイル server.c の中のステートレスのファイルサービス用サーバソースプログラムの概観 */
#include<stdio.h>
#include<rpc/rpc.h>
#include "StatelessFs.h"

/* READ PROCEDURE */
Data  *read_1(args)/* 単一の引数としてパラメータを入れる */
      readargs         *args;
{
      static Data results;/* static として宣言されなければならない */

      /* 位置 args.position で始まるファイル args.filename から arg.n バイト読み、そして
&result.buffer に読み込んだデータと result.n の読み込んだ実際のバイト数を出力
      する文 */

      return(&result);/* 単一の引数として結果を戻す */
}

/* WRITE PROCEDURE */
Nbytes  *write_1(args)/* 単一の引数としてパラメータを入れる */
       writeargs         *args;
{
       static Nbytes result;/* static として宣言されなければならない */

       /* 位置 args.position で開始するファイル args.filename にバッファ
      &args.data.buffer から args.data.n バイトを書き込むための文 */


      /* result に書き込まれた実際のバイト数を出すための文 */

      return(&result);
}
```

図 4.19　図 4.17 のステートレスのファイルサービスのサーバソースプログラムの概略

1. アプリケーションプログラマは、アプリケーションとしてクライアントプログラムとサーバプログラムを手で記述する。ここでの事例であるステートレスファイルサーバの2つのプログラムの基本的な部分を図4.18と4.19にそれぞれ示す。これらの2つのプログラムで使用される遠隔手続き名は、スタッブ手続き (read_1 と write_1) の名前である。

2. クライアントプログラムファイルがコンパイルされ、クライアントオブジェクトファイルが生成される。

3. サーバプログラムファイルがコンパイルされ、サーバオブジェクトファイルが生成される。

4. クライアントスタッブファイルとXDRフィルタファイルがコンパイルされ、クライアントスタッブオブジェクトファイルが生成される。

5. サーバスタッブファイルとXDRフィルタファイルがコンパイルされ、サーバスタッブオブジェクトファイルが生成される。

6. クライアントオブジェクトファイル、クライアントスタッブオブジェクトファイル及びクライアントのRPC実行時ライブラリがリンクされ、クライアントの実行ファイルが生成される。

7. サーバオブジェクトファイル、サーバスタッブオブジェクトファイル及びサーバのRPC実行時ライブラリがリンクされ、サーバの実行ファイルが生成される。

全体の処理概要を図4.20に示す。

手続きパラメータ

Sun RPCでは、遠隔手続きは1つのパラメータ構造を受理し、1つの結果を返す。したがって、入力又は出力として複数のパラメータを要求する手続きは、複数のパラメータを含む単一の構造を構成しなければならない。これは、構造データ (READ手続きに単一の出力パラメータ構造として使用される)、

readargs (READ手続きに単一の入力パラメータ構造として使用される) 及び writeargs (WRITE手続きに単一の入力パラメータ構造として使用される)

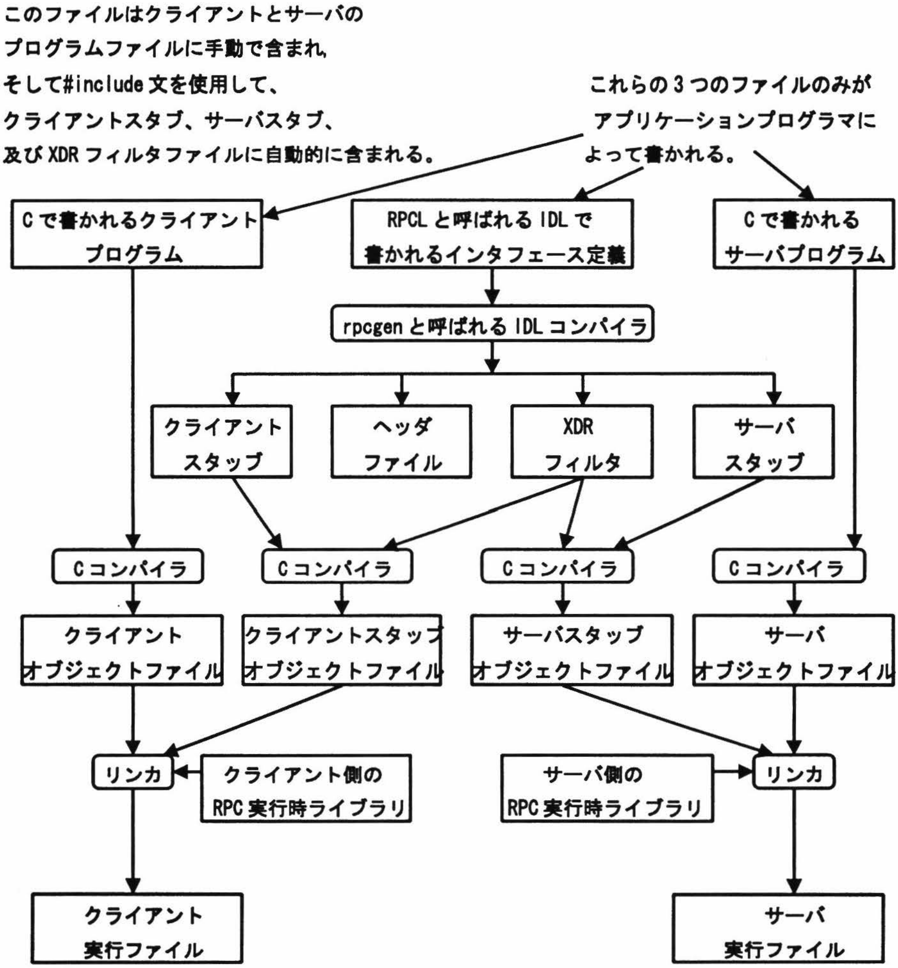


図 **4.20**　Sun RPC の RPC を作成の流れ

が図 4.17 から 4.19 の例で示すように定義されているためである。遠隔手続きがパラメータを持たない場合、遠隔手続きに対して NULL ポインタがパラメータとして渡されなければならない。したがって、Sun RPC 呼び出しは、必ず 2 つのパラメータ構造、1 つが遠隔手続きの単一パラメータ構造へのポインタともう 1 つがクライアントハンドルへのポインタを持つ (図 4.18 の read_1 と write_1 の呼び出しを参照)。一方、手続きが返すパラメータ構造は、単一な結果へのポインタである。返される結果は、手続きが返す時返される値は未定義であるため、サーバプログラム内の静的な変数として定義されなければならない (図 4.19 の返されるパラメータ構造 result を参照)。

パラメータと結果の結集

Sun RPC はパラメータと結果を渡すために固有のデータ構造を持つことは既に説明した。クライアントコンピュータとサーバコンピュータ間にデータ表現形式の差異が存在するため、これらのデータ構造は外部データ表現形式 (XDR) に変換され、結集手続きで使用される。使用される結集手続きは、ユーザによって規定され、RPC 実行時ライブラリの組み込み手続き若しくは組み込み手続きとして定義されるユーザ定義手続きである。RPC 実行時ライブラリは、全てのサイズ、文字、文字列、実数及びその他の型の整数を結集するための手続きを持つ。

XDR 符号化と復号化は同じアーキテクチャのクライアントとサーバ間でさえも必ず発生するため、XDR 符号化と復号化を必要としないアプリケーションのネットワークサービスに対して不必要なオーバヘッドが追加される。このような場合、ユーザ定義の結集手続きが利用できる。すなわち、ユーザは固有の結集手続きを書くことで、クライアントとサーバマシンのアーキテクチャが同じかを調べ、そうであれば変換を行わないでデータの処理ができる。アーキテクチャが同じでない場合には、XDR 手続きを利用することができる。

呼び出しセマンティクス

Sun RPC は最低 1 回セマンティクスをサポートしている。要求メッセージを送信後、RPC 実行時ライブラリは要求を再送する前にタイムアウト期間応答を待つ。再送回数は、待つ全体の時間をタイムアウト時間で除算して求められる。待つ全体の時間とタイムアウト時間は、それぞれ既定値として 25 秒と 5 秒である。これらの値はユーザによって変更できる。サーバから最終的に応答がなかった場合、RPC 実行時ライブラリはタイムアウト誤りを返す。

クライアントとサーバの結合

Sun RPCは、クライアントとサーバの結合のためのネットワークに適用できる結合サービスを提供しない。しかし、各ノードは全てのローカルサービス(既に述べたように、各サービスはプログラム番号と版番号で識別される)とポート番号に対応付けられるデータベースを管理するportmapperと呼ばれるローカル結合エージェントを持つ。このportmapperは、全てのノードの既知のポート番号で動作する。

サーバが動作を開始すると、プログラム番号、版番号及びポート番号をローカルのportmapperに登録する。クライアントがRPCを要求すると、遠隔手続きをサポートするサーバのポート番号を探し出す。このため、クライアントはプログラム番号と版番号(図4.18のclnt_create部を参照)を指定し、サーバホストのportmapperに遠隔要求を行う。これは、クライアントがサービスインタフェースをインポートする時にサーバのホスト名を指定しなければならないことを意味する。すなわち、これは、Sun RPCは位置透過性を持っていないことを意味している。

手続きclnt_createは、サービスインタフェースをインポートするためにクライアントによって使用される。これは、対応するサーバポート、例えばソケット記述子とソケットアドレスによって通信するために必要となる情報を含むクライアントハンドルを返す。クライアントハンドルは、サーバインタフェースの手続きに引き続いてRPCを行う場合、直接サーバと通信を行うためにクライアントによって使用される(図4.18のread_1とwrite_1の手続きを行うRPCを参照)。

例外処理

Sun RPCのRPC実行時ライブラリは、検出した誤りを処理するための幾つかの手続きを持っている。サーバ側の誤り処理手続きは、検出した誤りを伝えるためにクライアント側に応答メッセージを返送する。しかしながら、クライアント側の誤り処理手続きは、誤り報告機構を選択するための柔軟性を提供する。すなわち、誤りは、stderrに誤りメッセージを印刷するか、又はクライアントに誤りメッセージを返すことによってユーザに報告する。

セキュリティ

Sun RPCは、次の2つの認証をサポートする(フレイバとして参照される)。

1. **無認証 (no authentication)**。これは既定値である。この場合、要求された手続きを実行する前にクライアントの認証の検査をサーバは何も行わない。したがって、クライアントは要求メッセージに認証に関するパラメータを渡さない。

2. **UNIX 様式の認証 (UNIX-style authentication)**。この様式は、ユーザのある集合に対するサービスへのアクセスを制限させるために使用される。この場合、クライアントプログラムで実行するユーザの uid と gid が全ての要求メッセージで渡され、この認証情報に基づいてサーバは要求された手続きを実行すべきか否かを決定する。

3. **DES 様式の認証 (DES-style authentication)**。データ暗号化標準 (DES) は第 11 章で述べるように暗号化の技術である。DES 様式の認証では、各ユーザは netname と呼ばれる全体でユニークな名前をも持つ。クライアントプログラムで実行するユーザの netname が、全ての要求メッセージとして暗号化されて渡される。サーバ側では、暗号化された netname を復号化し、netname の情報から要求された手続きを実行するか否かを決定する。

DES 様式の認証は、UNIX 様式の認証よりもセキュリティを要求するユーザが使用するように推奨されている。DES 様式の認証を使用する RPC は、セキュア **RPC (secure RPC)** とも呼ばれる。

クライアントは RPC において上記の 3 つの認証のフレイバを選択できる柔軟性を持っている。認証のタイプは、クライアントのハンドルが生成される時に指定できる。RPC を行う前に要求するフレイバに認証タイプを設定することによって分散アプリケーション内でそれぞれの遠隔手続きに対して異なる認証機構を使用することができる。

Sun RPC の認証機構は、上記の 3 つの認証タイプに加えてユーザが自由に別の新しい認証タイプを使用できる点でオープンであると言える。

特別な RPC

Sun RPC は、非同期 RPC、コールバック RPC、ブロードキャスト RPC 及びバッチモード RPC をサポートしている。

非同期 RPC は、RPC のタイムアウト値を 0 に設定し、サーバからの応答を待たないことによって達成される。

コールバック RPC を実現するために、クライアントは転送するプログラム番号を使用するコールバックサービスをローカルの portmapper に登録する。プログラム番号は、RPC 要求の一部としてサーバに送られる。サーバは、コールバック RPC が使用できる状態になると与えられたプログラム番号を使用するクライアントに通常の RPC 要求を起動する。

ブロードキャスト RPC は全てのノードの portmapper に渡される。各ノードの portmapper は与えられたプログラム名とともにローカルサービスにそれを渡す。クライアントはそれぞれからの応答を受理する。

バッチモード RPC は、応答を要求しないで TCP/IP によってサーバにパイプラインのようにクライアントの呼び出しをバッチで送信することで行われる。

Sun RPC への批評

多いに普及していることから、Sun RPC に対して次のような批評がされている。

1. Sun RPC は、クライアントがサービスインタフェースをインポートする場合に、サーバのホスト名を指定しなければならないため位置透過性を持っていない。

2. Sun RPC のインタフェース定義言語は、手続きのパラメータと結果について汎用的な仕様になっていない。これは、複数のパラメータや結果を単一の構造にすることを要求するものである。

3. Sun RPC は、トランスポートに独立でなく、トランスポートプロトコルは UDP や TCP に限定されてる。しかしながら、TI-RPC(トランスポート非依存 RPC) と呼ばれる Sun RPC のトランスポート非依存な版が Sun Soft 社から開発されている。TI-RPC は、ユーザの参照やトランスポートの可動性に従って動的に選択される単純でかつ一貫性のある方式を提供する。

4. UDP において、Sun RPC メッセージ長は 8 キロバイト以下に制限される。

5. Sun RPC は、あるアプリケーションにとっては使用できない最低 1 回セマンティクスのみをサポートする。

6. Sun RPC は、ネットワークに適用できるクライアントとサーバの結合サービスを提供していない。

7. 既に 4.18 節で、RPC に基づくアプリケーションの性能向上のためにクライアント又はサーバプロセスの実装にスレッドが使用できることを説明した。Sun OS は個別のスレッドパッケージを持っているが、Sun RPC はクライアン

トやサーバにおいてスレッドを統合化する機能を提供していない。

4.20.2 DCE RPC

DCE RPC は、DCE の全ての通信の基本となっているため、DCE の最も基盤的な構成要素の 1 つある。これは、Apollo 社 (現在は Hewlett ー Packard 社) によって開発された Network Computing System(NCS) から派生している。

DCE RPC も自動スタッブ生成方式である。アプリケーションのインタフェース定義は IDL で書かれる。例えば、図 4.17 のステートレスのファイルサーバのインタフェース定義は、IDL で図 4.21 のように書き直される。Sun RPC とは異なり、DCE RPC IDL は手続きのパラメータと結果について完全で汎用的な仕様を与える。図に示すように、各インタフェースは、16 進化 10 進法で ASCII 文字列として IDL ファイル内に 128 ビットの 2 進数であるユニバーサリ固有識別子 (UUID) によって一意に識別される。各 UUID の一意性は、生成時のタイムスタンプと場所の情報によって保証される。インタフェース定義のテンプレートと同様に UUID は、unidgen ユーティリティによって作成される。したがって、サービスのインタフェース定義を生成するために、最初のステップとして unidgen プログラムが呼ばれる。自動的に生成されるテンプレートファイルは、定数やサービスの手続きインタフェースを定義するために手動で編集される。

IDL ファイルをコンパイルする場合、クライアントとサーバスタッブ及びヘッダファイルを生成するために IDL コンパイラを使用する。クライアントとサーバプログラムは最終的にはアプリケーションとして手で書かれ、図 4.20 と同じステップがクライアントとサーバの実行ファイルを生成するために取られる。

DCE RPC の遠隔手続きの既定値の呼び出しセマンティクスは、最大 1 回セマンティクスである。しかしながら、繰り返し等価の性質を持つ手続きに対してはこのような奇妙な呼び出しセマンティクスは必要ない、したがって、DCE RPC は、繰り返し等価な手続きの IDL 定義の一部として示しことでアプリケーションプログラマに柔軟性を提供する。この場合、誤り回復は最大 1 回セマンティクスで使用される複雑なプロトコルよりもむしろ単純な再送方式によって行われる。

DCE RPC は、ディレクトリサービスに基づくクライアントとサーバの結合に対してネットワークに適用できる結合サービスを持つ (DCE ディレクトリサービスの詳細は第 10 章で説明する)。ここでは、DCE システムの全てのセル

```
[uuid (b20a1705-3c26-12d8-8ea3-04163a0dcefz)
 version (1.0)]

interface stateless_fs
{
          const long FILE_NAME_SIZE = 16;
          const long BUFFER_SIZE =1024;
          typedef char FileName{File_Name_SIZE};
          typedef char Buffer{BUFFER_SIZE};

          void read(
                    [in] FileName          filename;
                    [in] long               position;
                    [in, out] long           nbytes;
                    [out] Buffer            buffer;
          );
          void write(
                    [in] FileName          filename;
                    [in] long               position;
                    [in, out] long           nbytes;
                    [out] Buffer            buffer;
          );
}
```

図 4.21 DCE RPC の IDL で書かれた図 4.17 のステートレスのファイルサービスのインタフェース定義

は、セル内で使用される名前付け環境を制御するセルディレクトリサービス(CDS)と呼ばれる構成要素を持つことを言及することにとどめる。さらに、各DCEサーバノードでは、(サーバ、エンドポイントの)エントリのデータベースを管理するrpcd(RPCデーモン)と呼ばれるデーモンプロセスが動作する。エンドポイントは、マシン上のサーバのプロセスアドレス(例えばTCP/IPポート番号)である。

アプリケーションサーバが初期化するとき、オペレーティングシステムにエンドポイントを問い合わせる。そして、ローカルrpcdにエンドポイントを登録する。初期化のときに、サーバはセルのCDSとともにホストアドレスも登録する。

クライアントがサーバを呼び出すための最初のRPCを行うと、クライアントスタッブは、サーバが実行するホストを見つけるために要求を出すCDSのサーバと交信することによってサーバのホストアドレスを得る。そして、サーバのエンドポイントを得るためにサーバのホストのrpcd(rpcdは全てのホストの既知のエンドポイントを持つ)と交信する。サーバのエンドポイントが分かるとRPCが行われる。同じサーバに対して行われる以降のRPCでは、この検索処理は必要ではない。

上で述べたようなステップは、クライアントとサーバが同じセルに属している場合にはクライアントとサーバの結合として使用される。クライアントは、別のセルに属するサーバとRPCを行うことができる。この場合、サーバのホストアドレスを得るための処理は、セル外(セル間)のグローバルな名前付け環境を制御できるグローバルディレクトリサービス(GDS)を呼び出す(詳細は第10章を参照)。

DCE RPCは、ブロードキャスト機能を提供する。この機能を使用するために、遠隔手続きは定義するIDLファイルのブロードキャスト属性を持たなければならない。この属性を持つ手続きが呼び出されると、要求は要求されたインタフェースにある全てのサーバに送信される。この要求を受信した全てのサーバは応答し、最初の応答のみが呼び出し側に返され、その他の応答はRPC実行時ライブラリで廃棄される。

4.21 まとめ

遠隔手続き呼び出し(RPC)は、分散コンピューティングシステムにおいて幅広く受け入れられるIPC機構であるIPCの汎用的なメッセージパッシング

モデルの特殊なケースである。この普及は、単純な呼び出し形式、親しみ易い手続き呼び出しセマンティクス、利用のし易さ、汎用性、有効性及び良く知られたインタフェース仕様によるところが大きい。RPC の理想的な透過性とは、遠隔手続き呼び出しはローカル手続き呼び出しとの差異がないことである。しかしながら、このことは通常部分的にしか達成されない。

RPC 機構の実装において、プログラムは 5 つの部分、クライアント、クライアントスタッブ、RPC 実行時ルーチン、サーバスタッブ、及びサーバから構成される。クライアントとサーバスタッブは、呼び出しメッセージ又は応答メッセージに含まれるデータの処理を行うことにより、ネットワークでの転送や受信プロセスに適したデータになる。一方、RPC 実行時ルーチンは透過的な方法でネットワークサービスを提供する。

RPC システムには、2 つのメッセージ、呼び出しメッセージと応答メッセージがある。呼び出しメッセージは、遠隔手続きの実行を要求するためにクライアントからサーバに送信され、応答メッセージは遠隔手続きの実行結果としてサーバからクライアントに送られる。RPC メッセージのデータの符号化と復号化の処理は、結集と呼ばれる。

RPC に基づくアプリケーションのサーバは、ステートフル又はステートレスである。さらに、RPC サーバがサービスする時間間隔に依存して、サーバは 3 つのタイプ、呼び出しインスタンスサーバ、トランザクション／セッションインスタンスサーバ及び永続サーバがある。これらのサーバの選択は、設計するアプリケーションの必要性に依存する。

RPC 機構を設計する上でパラメータパッシングセマンティクスとして、値パラメータと参照パラメータがある。多くの RPC 機構は、クライアントとサーバが異なるアドレス空間に存在するために値パラメータセマンティクスを使用する。

RPC 機構の呼び出しセマンティクスは、遠隔手続きが障害状態でどのくらい実行されるかを決めるものである。RPC 機構での呼び出しセマンティクスとして、最低 1 回、最大 1 回、ちょうど 1 回の 3 つのタイプがある。もちろん、ちょうど 1 回セマンティクスが強力で望ましいものである。

IPC の要求に基づいて、システムは RPC の通信プロトコルとして、要求 (R) プロトコル、要求／応答 (RR) プロトコル及び要求／応答／確認通知 (RRA) プロトコルのいずれかを使用する。これらに加えて、長い呼び出し処理や呼び出し相互の間が長いもの、又は単一のデータグラムパケットにパラメータや結果が入らないような複雑な RPC を処理するために特殊な通信プロトコルが使

用される。

クライアントとサーバの結合は、遠隔手続き呼び出しを行うために必要となる。結合のための汎用的なモデルは、サーバがサービス提供ためにその意向を登録するために動作をエクスポートし、クライアントはサービスが必要になった時に動作をインポートすることである。クライアントは、コンパイル時、リンク時又は実行時にサーバと結合できる。

RPC の特殊のタイプが通常の RPC プロトコルとは異なった方法で行われる。例えば、非同期 RPC はクライアントからサーバへの 1 方向のメッセージ転送を、コールバック RPC は参加しているプロセス内の中でクライアントとサーバの関係とは異なったピアツーピア通信を提供し、ブロードキャスト RPC は 1 対 1 通信に代わってブロードキャスト及びマルチキャスト通信の機能を、バッチモード RPC は通常の同期通信とは違って非同期通信モードのクライアント要求のバッチ転送機能を提供する。

RPC が設計された時に実装がほとんど決まっていた従来の RPC システムとは異なり、トランスポートプロトコル、データ表現及び制御プロトコルは、不均質な環境に向けて設計された RPC システムでは結合する時に何が使用されるかが決まる。このために、このような RPC システムで使用される結合は、従来の RPC システムでの結合よりもより多くの情報によって行われる。

Bershad 等 [1990] は、マイクロカーネルのオペレーティングシステムでドメイン間通信を行うために設計され、最適化された通信機能を持つライトウエイト遠隔手続き呼び出しの使用を提案している。従来の RPC システムよりも性能を向上するために、LRPC では 4 つの技術、単純の制御転送、単純なデータ転送、単純なスタッブ及び並行性のための設計を使用している。

RPC 機能を使用する分散アプリケーションの性能を向上するための最適化は、複数サーバへの並行アクセス、複数の同時要求の処理、呼び出し当たりのサーバ負荷の軽減、繰り返し等価な遠隔手続きの応答のキャッシング、タイムアウト値の適切な選択、及び RPC プロトコルの適切な設計である。

4.22 演習問題

4.1 RPC 機能の開発の主な動機は何か？ RPC 機能は分散アプリケーションプログラマの仕事をどのように簡単にするか？

4.2 RPC モデルと従来の手続き呼び出しモデルとの類似点と相違点は何か？

4.3 従来に手続き呼び出しモデルでは、呼び出し側と呼ばれる側の手続きは

お互いに通信するためにグローバル変数を使用している。RPC モデルではこのようなグローバル変数が何故使用されないかを説明せよ。

4.4 RPC では、呼ばれる手続きは呼び出す手続きと同じコンピュータ若しくは別のコンピュータに存在する。呼ばれる手続きと呼び出す手続きが同じコンピュータに存在する場合でも、何故遠隔手続き呼び出しの用語が使用されるかを説明せよ。

4.5 透過的な RPC 機構を設計する上での主な課題は何か？ RPC 機構の完全な透過性を達成することは可能か？ どちらの場合についてもその理由を説明せよ。

4.6 呼び出しと呼ばれる手続きが異なるコンピュータに存在する場合に RPC 機構の完全な透過性を達成することは、2 つのプロセスの間にメッセージ通信のためにネットワークが介在するため不可能に近い。呼び出しと呼ばれる手続きが同じコンピュータに存在する RPC 機構を想定した場合、この RPC 機構の完全な透過性は可能か？理由も合わせて示せ。

4.7 スタッブとは何か？ スタッブはどのように生成されるか？ RPC 機構の透過性を実現するためにスタッブの役割を説明せよ。

4.8 サーバが簡単な整数演算 (加算、減算、乗算、除算) を行い、クライアントは RPC 機構を使ってサーバと相互作用を持つとする。各構成要素の目的を説明し、かつこの RPC アプリケーションの呼び出し及び応答メッセージの内容を示せ。誤りの場合、例えば 0 での除算又はオーバフローの発生の時、サーバはクライアントに誤りの内容を適切に通知しなければならない。

4.9 図 4.8 の RPC アプリケーションのメッセージの内容を結集するためにタグと非タグ表現の両者の結集の手続きを記述せよ。

4.10 ユーザ定義のプログラムオブジェクトが次のような順のデータ型から構成される構造を持つとする。ブール型、整数型、長整数型、及び 8 文字の固定長の文字列型。このプログラムオブジェクトに対するタグ及び非タブの両者の結集手続きを記述せよ。RPC ソフトウェアは基本のデータ型の結集を提供するものとする。

4.11 RPC の呼び出しプロセスは、呼び出しを行った後、呼ばれたプロセスからの応答を待たなければならない。これは実際どのように行われるかを説明せよ。

4.12 ステートフルとステートレスのサーバには相違がある。ステートフルサーバはプログラミングの容易性やステートレスサーバよりも一般的に効果的である言われるが、分散アプリケーションでステートレスサーバを何故使用す

るか？

4.13 下記のアプリケーションに対して最適なサーバ生成セマンティクスを示せ。

(a) サービスは1度のみ要求され、クライアントがこのサービスのサーバと相互作用を行うセッションでは、1つの呼び出しと1つの応答メッセージをクライアントとサーバ間で交信する。

(b) サービスは1度のみ要求され、クライアントがこのサービスのサーバと相互作用を行うセッションでは、複数の呼び出しと複数の応答メッセージをクライアントとサーバ間で交信する。

(c) サーバは複数のクライアントの要求をサービスする。

4.14 サーバは複数のクライアントに共有される。異なるクライアントからの並行的な要求が相互に独立するように処理するためのサーバの遠隔手続きの設計方式を述べよ。

4.15 多くのRPCシステムは、何故パラメータパッシングとして値パラメータセマンティクスをサポートするか？

4.16 オブジェクトに基づくシステムで使用される次のパラメータパッシングセマンティクス間の類似性と相違を述べよ。

(a) オブジェクト参照パラメータ

(b) 移動パラメータ

(c) 滞在パラメータ

4.17 RPCセマンティクスは、何故従来の手続き呼び出しとは違うかを説明せよ。また、不確実、最後1回、最後、最低1回、及びちょうど1回のそれぞれの呼び出しセマンティクスの相違を分類せよ。これらの実装をどのように行うかを説明せよ。

4.18 孤児呼び出しとは何か？ 次のような呼び出しセマンティクスの実装において孤児呼び出しはどのように処理されるか？

(a) 最後1回呼び出しセマンティクス

(b) 最後呼び出しセマンティクス

(c) 最低1回呼び出しセマンティクス

4.19 次のアプリケーションに対して、不確実、最後1回、最後、最低1回、若しくはちょうど1回呼び出しセマンティクスのどれが使用されるべきかを理由とともに示せ。

(a) 時刻を知るための時間サーバへの要求

(b) ノードの資源利用可能状況を知るためのノードの資源管理への要求

(c) 全てのノードのプロセス管理が手動でシステム全体の負荷を分担しているシステムにおいて、プロセス管理がノードの現在のジョブの数を定期的にブロードキャストする。

(d) 式の計算を行うために計算するサーバへの要求

(e) 席の空き状況を知るための予約サーバへの要求

(f) 席を予約するための予約サーバへの要求

(g) 指定する位置に読み書きポインタを設定するためのファイルサーバへの要求

(h) 既存のファイルにレコードの追加をするためのファイルサーバへの要求

(i) オブジェクトの移動性をサポートしないシステムにおいて、名前付けされたオブジェクトの位置を知るための名前サーバへの要求

(j) オブジェクトの移動性をサポートするシステムにおいて、名前付けされたオブジェクトの位置を知るための名前サーバへの要求

4.20 多くの RPC システムが確認通知メッセージを使用しない理由を説明せよ。RPC での R、RR、及び RRA プロトコルの相違を示せ。各プロトコルに対して最適なアプリケーションの例を示せ。

4.21 RPC 要求をサーバで処理するための時間を T(T は大変大きい) とする。RPC 要求を行うクライアントが時間 T 後にサーバから応答を受信すると知っていても、クライアントとサーバ間の通信リンクの障害やサーバのクラッシュのためにサーバに要求が届かないことによってサーバからの応答を不必要にクライアントが待つことになる。この状況を避けるための機構を示せ。すなわち、このような異常状態をクライアントが検出し、できるだけ早く正しい処置を行えるべきである。

4.22 クライアントとファイルサーバ間で頻繁に大量のデータが転送されるような RPC 機構を設計するとする。しかしながら、下位のネットワークは最大パケット長が 4 キロバイトに制限されている。4 キロバイトのパケットが 4 ms で転送でき、ヌル RPC(すなわち、0 データバイト) を行うために必要な時間が 0.5 ms とする。各 RPC 要求に平均 16 キロバイトのデータを転送する場合、次の 2 つの方法は設計としてどちらが良いか？

(a)1 つの論理的な RPC に対して複数の物理的な RPC を使用し、各物理的な RPC は単一のパケットデータを転送する。

(b) 単一 RPC はマルチデータグラムメッセージとしてデータを転送する。

4.23 クライアントとサーバ間の結合を動的に変更する RPC システムの主要な利点は何か？ この柔軟性を提供する上での課題は何か？ 示した課題の処理

方法を説明せよ。

4.24 サーバは複数のクライアントにサービスを同時に提供するように設計される。クライアントが複数のサーバと同時に相互作用を持つことはあるか？ ない場合には理由を、ある場合には例を 2 つ示せ。

4.25 コンパイル時、リンク時及び呼び出し時のクライアントとサーバの結合方法の利点と欠点を述べよ。

4.26 サーバのインタフェース名が与えられた時、サーバの位置を知るためにブロードキャスト方式と名前サーバによる方式の利点と欠点を述べよ。

4.27 コールバック RPC 機能と何か？ この機能が有効となるアプリケーションの例を示せ。RPC システムでこの機能をサポートする場合に主な課題は何か？ 示した課題の処理方法を説明せよ。

4.28 次の機能が有効となるアプリケーションの例を示せ。

(a) ブロードキャスト RPC 機能

(b) マルチキャスト RPC 機能

これらの機能を実現するための機構を説明せよ。

4.29 バッチモード RPC 機能が有効となるアプリケーションの特徴を示せ。この機能を使用する場合の主な問題は何か？

4.30 HRPC システムのクライアントとサーバの結合機構 [Bershad et al. 1987] の詳細を示し、このシステムで結合が行われるまでにトランスポートプロトコル、データ表現、及び制御プロトコルの選定がどのように行われるかを説明せよ。

4.31 ライトウエイト RPC(LRPC) システム [Bershad et al. 1990] の開発の主な動機は何か？従来の RPC システムと比較して LRPC システムの効果的な技術を説明せよ。

4.32 単純な RPC 機構を使用して実現されたクライアントサーバモデルにおいて、RPC 要求を行った後、クライアントはその要求の応答をサーバから受信するまで待ち続ける。サーバが要求を処理している間 (特に処理時間が長い場合) クライアントが別の処理を行うことは有効である。このような場合、クライアントが別の処理を行うための 3 つの機構を説明せよ。

4.33 クライアントサーバモデルは、RPC 機構を使って実現されるべきである。サーバは多くのクライアントの要求をサービスするために共有サーバは RPC 機構における高価な資源として実現されている。RPC 機構全体の性能を向上するためにこのような共有サーバをどのように設計すべきかの指針を示せ。

参考文献

[Almes 1986] Almes, G. T., "the Impact of Language and System on Remote Procedure Call Design," In: *Proceedings of the 6th International Conference on Distributed Computing Systems*, IEEE Press, Piscataway, NJ, pp. 414-421 (May 1986).

[Almes et al. 1985] Almes, G. T., Black, A. P., Lazowska, E. D., and Noe, J. D., "The Eden System: A Technical Review," *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 1, pp. 43-59 (1985).

[Bacon and Hamilton 1987] Bacon, J. M., and Hamilton, K. G., "Distributed Computing with RPC: The Cambridge Approach," Technical Report No. 117, Computer Laboratory, University of Cambridge, England (1987).

[Bal et al. 1987] Bal, H. E., Renesse, R., and Tanenbaum, A. S., "Implementing Distributed Algorithms Using Remote Procedure Calls," In: *Proceedings of the AFIPS National Computer Conference*, Chicago, IL, pp. 499-506 (June 1987).

[Bal et al. 1989] Bal, H. E., Steiner, J. G., and Tanenbaum, A. S., "Programming Languages for Distributed Computing Systems," *ACM Computing Surveys*, Vol. 21, No. 3, pp. 261-322 (1989).

[Bershad et al. 1987] Bershad, B. N., Ching, D. T., Lazowska, E. D., Sanislo, J., and Schwartz, M., "A Remote Procedure Call Facility for Interconnecting Heterogeneous Computer Systems," *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 8, pp. 880-894 (1987).

[Bershad et al. 1990] Bershad, B. N., Anderson, T. E., Lazowska, E.D., and Levy, H. M., "Lightweight Remote Procedure Call," *ACM Transactions on Computer Systemss*, Vol. 8, No. 1, pp. 37-55(1990). ©ACM, Inc., 1990.

[Birrell 1985] Birrell, A. D., "Secure Communication Using Remote Procedure Calls," *ACM Transactions on Computer Systems*, Vol. 3, No. 1, pp. 1-14 (1985).

[Birrell and Nelson 1984] Birrell, A. D., and Nelson, B., "Implementing Remote Procedure Calls," *ACM Transactions on Computer Systems*, Vol.2, No. 1, pp. 39-59 (1984).

[Birrell et al. 1982] Birrell, A. D., Levin, R., Needham, R. M., and Schroeder, M. D., "Grapevine: An Exercise in Distributed Computing," *Communications of the ACM*, Vol. 25, No. 4, pp.260-274 (1982).

[Black et al. 1986] Black, A., Hutchinson, N., Jul, E., and Levy, H., "Object Structure in the Emerald System." In: *Proceedings of the 1st ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA-1986)*, pp. 78-86 (1986).

[Black et al. 1987] Black, A., Hutchinson, N., Jul, E., Levey H., and Carter, L., "Distribution and Abstract Types in Emerald," *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 1, pp.65-76 (1987).

[Corbin 1991] Corbin, J. R., The Art of Distributed Applications, Springer-Berlag, New York, NY (1991).

[Davison et al. 1992] Davison, A., Drake, K., Roberts, W., and Slater, M., *Distributed Window Systems, A Practical Guide to X11 and Open-Windows*, Addison-Wesley, Reading, MA (1992).

[DES 1977] DATA ENCRYPTION STANDARD, *FIPS Publication 46*, National Bureau of Standards, U.S. Department of Commerce, Washington DC (January 1977).

[Gibbons 1987] Gibbons, P. B., "A Stub Generator for Multi-language RPC in Heterogeneous Environment," *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 1, pp. 77-87 (1987).

[Gifford and Glasser 1988] Gifford, D., and Glasser, N., "Remote Pipes and Procedures for Efficient Distributed Communication," *ACM Transactions on Computer Systems*, Vol. 6, No. 3, pp. 258-283 (1988).

[Gimson 1985] Gimson, R., "Call Buffering Service," Technical Report No. 19, Programming Research Group, Oxford University, Oxford, England (1985).

[Goscinski 1991] Goscinski, A., "Distributed Operating Systems, The Logical Design," Addison-Wesley, Reading, MA (1991).

[Hamilton 1984] Hamilton, K. G., "A Remote Procedure Call System," Ph.D. Dissertation, Technical Report No. 70, Computer Laboratory, University of Cambridge, England (December 1984).

[Harbinson 1992] Harbinson, S. P., *Modula-3*, Prentice-Hall, Englewood Cliffs, NJ (1992).

[Hiltunen and Schlichting 1995] Hiltunen, M. A., and Schlichting, R. D., "Constructing a Configurable Group RPC Service," In: *Proceedings of the 15thInternational Conference on Distributed Computing Systems*, IEEE Press, Piscataway, NJ (May 1995).

[Hutchinson et al. 1989] Hutchinson, N. C., Peterson, L. L., Abbott, M. B., and O'Malley, s., "RPC in the x-Kernel: Evaluating New Design Techniques," In: *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pp. 91-101 (1989).

[Jones et al. 1985] Jones, M. B., Rashid, R. F., and Thompson, M. R., "Matchmaker: An Interface Specification Language for Distributed Processing," In: *Proceedings of the 12th ACM Symposium on Principles of Programming Languages*, pp. 225-235(1985).

[Karger 1989] Karger, P. A., "Using Registers to Optimize Cross-Domain Call Performance," In: *Proceedings of the 3rd Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 194-204 (April 1989).

[Khanna 1994] Khanna, R. (Ed.), *Distributed Computing: Implementation and Management Strategies*, Prentice-Hall, Englewood Cliffs, NJ (1994).

[Kim and Purtilo 1995] Kim, T. H., and Purtilo, J. M., "Configuration-Level Optimization of RPC-Based Distributed Programs," In: *Proceedings of the 15th International Conference on Distributed Computing Systems*, IEEE Press, Piscataway, NJ (May 1995).

[Lin and Gannon 1985] Lin, K. J., and Gannon, J. D., "Atomic Remote Procedure Call," *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 10, pp. 1126-1135 (1985).

[Liskov et al. 1981] Liskov, B., Moss, E., Schaffert, C., Sheifler, R., and Snyder, A., "CLU Reference Manual," In: *Lecture Notes in Computer Science 114*, Springer-Verlag, Berlin (1981).

[Liskov and Scheifler 1983] Liskov, B., and Scheifler, R., "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," *ACM Transactions on Programming Languages and Systems*, Vol. 5, No. 3, pp. 381-404 (1983).

[Liskov and Shrira 1988] Liskov, B., and Shrira, L., "Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems," In: *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation*, Association for Computing Machinery, New York, NY, pp. 260-267 (June 1988).

[Lockhart Jr. 1994] Lockhart, Jr., H. W., *OSF DCE: Guide to Developing Distributed Applications*, IEEE Computer Society Press, Los Alamitos, CA (1994).

[Nelson 1991] Nelson, G. (Ed.), *Systems Programming with Modula-3*, Prentice-Hall, Englewood Cliffs, NJ (1991).

[Panzieri and Srivastava 1988] Panzieri, F., and Srivastava, S. K., "Rajdoot: A Remote Procedure Call Mechanism with Orphan Detection and Killing," *IEEE Transactions on Software Engineering*, Vol. SE-14, pp. 30-37 (1988).

[Rosenberry et al. 1992] Rosenberry, W., Kenney, D., and Fisher, G., *OSF DISTRIBUTED COMPUTING ENVIRONMENT, Understanding DCE*, O'Reilly & Associates, Sebastopol, CA (1992).

[Schroeder and Burrows 1990] Schroeder, M. D., and Burrows, M., "Performance of Firefly RPC," *ACM Transactions on Computer Systems*, Vol. 8, No. 1, pp. 1-17 (1990).

[Shirley et al. 1994] Shirley, J., Hu, W., and Magid, D., *Guide to Writing DCE Applications*, 2nd ed., O'Reilly & Associates, Sebastopol, CA (1994).

[Shrivastava et al. 1991] Shrivastava, S., Dixon, G. N., and Parrington, G. D., "An Overview of the Arjuna Programming System," *IEEE Software* (January 1991).

[Smith 1994] Smith, B., "Client/Server Made Easy," *BYTE*(March 1994).

[Spector 1982] Spector, A. Z., "Performing Remote Operations Efficiently on a Local Computer Network," *Communications of the ACM*, Vol. 25, No. 4, pp. 246-259 (1982).

[Srivastava and Panzieri 1982] Srivastava, S. K., and Panzieri, F., "The Design of Reliable Remote Procedure Call Mechanism," *IEEE Transactions on Computers*, Vol. C-31, No. 7 (1982).

[Sun Microsystems 1985] "Remote Procedure Call Protocol Specification," *Networking on the Sun Workstation*, Sun Microsystems, Mountain View, CA (1985).

[Sun Microsystems 1990] *Network Programming*, Sun Microsystems, Mountain View, CA (1990).

[Tanenbaum and Van Renesse 1988] tanenbaum, A. S., and Van Renesse, R., "A Critique of the Remote Procedure Call Paradigm," In: *Proceedings of the EUTECO'88 Conference*, Vienna Austria, North-Holland, Amsterdam, pp. 775-783 (April 1988).

[Tay and Ananda 1990] Tay, B. H., and Ananda, A. L.,"A Survey of Remote Procedure Calls," *Operating Systems Review*, Vol. 24, pp. 68-79 (1990).

[Walker et al. 1990] Walker, E. F., Floyd, R., and Neves, P., "Asynchronous Remote Operation Execution in Distributed Systems," In: *Proceedings of the 10th International Conference on Distributedd Computing Systems*, IEEE Press, Piscataway, NJ,pp.253-259(1990).

[Wilbur and Bacarisse 1987] Wilbur, S., and Bacarisse, B., "Building Distributed Systems with Remote Procedure Call," *Software Engineering Journal*, pp. 148-159 (September 1987).

[Xerox Corpration 1981] "Courier: The Remote Procedure coall Protocol Xerox system Integration Standard XSIS-038112, Stanford, CT(1981).

インターネット参考文献

遠隔手続き呼び出しに関する参考文献は見当たらないが、下記の参考文献がある。

ftp:ftp.cs.umanitoba.ca/pub/bibliographies/Os/IMMD_IV.html
ftp:ftp.cs.umanitoba.ca/pub/bibliographies/Os/os.html
ftp:ftp.cs.umanitoba.ca/pub/bibliographies//Misc/misc.1.html
ftp:ftp.cs.umanitoba.ca/pub/bibliographies/Parallel/JPDC.html

第5章

分散共有メモリ

5.1　はじめに

第3章では、プロセス間通信の2つの基本的なパラダイムとして、以下のものを説明した。

■共有メモリパラダイム
■メッセージパッシングパラダイム

メッセージパッシングシステム (第3章で述べた)、又は遠隔手続き呼び出し (RPC) をサポートしているシステム (第4章で述べた) は、メッセージパッシングパラダイムに分類される。このパラダイムは、プロセス間通信に2つの基本的なプリミティブからなる。

Send(recipient, data)
Receive(data)

送信プロセスは、共有すべきデータを生成してそれを通信したい受信側に送る。受信側では、データを受信する。この機能は、ときには言語レベルの構造で隠蔽される。例えば、RPCは手続きの仕様に基づいて自動的にメッセージを生成して受信する機能を提供する。しかし、通信するプロセスは共有するデータを交換するために直接相互作用をするため、基本的な通信パラダイムはメッセージパッシングシステムと同じである。

メッセージパッシングパラダイムと対称的に、共有メモリパラダイムはプロセスに共有アドレス空間を提供する。プロセスは、このアドレス空間を通常のローカルメモリを使うのと同じ方法で使うことができる。つまり、プロセスは共有メモリ空間を以下の 2 つの基本プリミティブを通してデータにアクセスする。もちろん、異なる実装には構文上のそして意味上のバリエーションがある。

data = Read(address)
Write(address, data)

Read は address で参照される data 項目を返す。そして、Write は address によって参照される内容を data の値に設定する。

第 1 章において、一般的になりそして商業的にも受け入れられるようになってきた**命令多重 (multiple-instruction)，データストリーム多重 (multiple-data-stream)(MIMD)** 型マルチプロセッサの 2 つが、密結合共有メモリマルチプロセッサと粗結合分散メモリマルチプロセッサであることを見てきた。プロセス間通信のための共有メモリパラダイムの利用は、密結合共有メモリマルチプロセッサにおいて動作している分散プロセスにとってごく自然である。しかし、粗結合分散メモリシステムにとっては、プロセス間通信のための共有メモリパラダイムをサポートするための物理的に共有されているメモリはない。したがって、最近まで粗結合の分散メモリマルチプロセッサにおけるプロセス間通信機構はメッセージパッシングパラダイムに限定されていた。しかし、幾つかの最近の粗結合分散メモリシステムは、メッセージパッシング通信システムの上に、プログラマに抽象的な共有メモリを提供するための 1 つのソフトウェア階層を実装してきた。共有メモリ抽象層は、この分散メモリシステムにメモリが物理的に共有されているような錯覚を与え、プログラマにその共有メモリパラダイムの利用を認めるものである。このソフトウェア階層は、共有メモリの抽象化を提供しているが、オペレーティングシステムカーネルの内部に実現されることも、適切なシステムカーネルのサポートを使った実行時ライブラリルーチンに実現されることもある。**分散共有メモリ (Distributed Shared Memory (DSM))** という用語は、粗結合の分散メモリシステムに共有メモリパラダイムが適用されたものを指している [Stumm and Zhou 1990]。

図 5.1 に示すように、DSM は粗結合プロセッサのプロセス間に共有された仮想アドレス空間を提供する。つまり、DSM は基本的に 1 つのネットワーク環境における異なるマシン上のローカルメモリを、複数のサイトで動作する協調プ

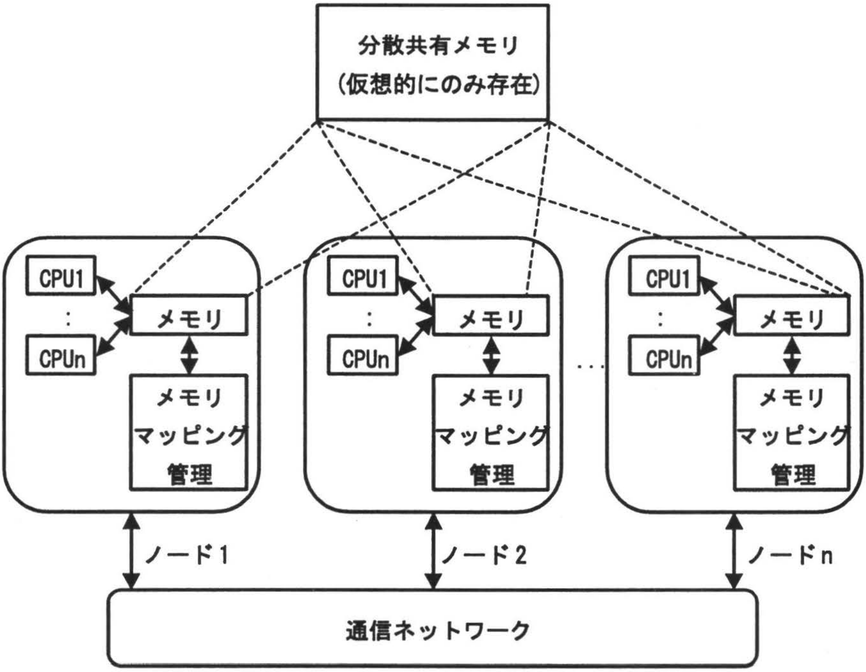


図 **5.1** 分散共有メモリ (DSM)

ロセスによって共有された1つの論理的な実体として統合している。この共有メモリ自体は、仮想的に存在するのみである。応用プログラムは、それを従来の仮想メモリと同様な方法で利用することができる。ただし、それを利用するプロセスは異なるマシン上で並列に動作できることが従来とは異なる。共有メモリの仮想的な存在のために、DSM は時々**分散共有仮想メモリ (Distributed Shared Virtual Memory (DSVM))** として呼ばれることもある。

5.2 DSM システムの一般的なアーキテクチャ

DSM システムは、普通は図 5.1 に示すようなアーキテクチャを持っている。システムの各ノードは、1つ以上の CPU と1つのメモリ装置からなっている。このノードは、高速通信ネットワークで接続されている。簡易なメッセージパッシングシステムによって、異なるノード上のプロセスは互いにメッセージ交換が可能となっている。

DSM 抽象は、すべてのノードのプロセッサに対して大きな共有メモリを提供する。密結合の並列アーキテクチャでの共有物理メモリと比較して、DSM の共有メモリは仮想的にのみ存在している。各ノードのソフトウェアによるメモリマッピング管理のルーチンがローカルメモリを仮想メモリへマッピングしている。マッピング作業を容易にするために、共有メモリ空間は、**ブロック (block)** に分割されている。

データキャッシングは、メモリアクセスの遅延に対応するよく知られた解である。データキャッシングの考え方は、DSM においてもネットワーク遅延を削減するために使われている。つまり、個々のノードの主メモリは共有メモリ空間のキャッシュピースとして使われる。各ノードのメモリマッピング管理者は、そのローカルメモリを関連するプロセッサ群の共有メモリ空間の大きなキャッシュとして見せている。キャッシュの基本的な単位はメモリブロックである。

あるノードのプロセスが、共有メモリ空間のメモリブロックからアクセスするとき、ローカルメモリマッピング管理者はその要求を引き受ける。もし、アクセスされたデータが含まれているメモリブロックがローカルメモリにある場合、その要求はアクセスされたデータをローカルメモリから提供することで満足される。また、ローカルメモリにない場合はネットワークブロックフォルトを発生し、制御はオペレーティングシステムに渡される。オペレーティングシステムは次に要求されたメモリブロックが存在するノードにそのブロックを獲得するためのメッセージを送る。不足していたブロックは、遠隔のノードからクライアントのプロセスのノードに移されて、オペレーティングシステムはそれをアプリケーションのアドレス空間に対応付ける。停止フォルトしていた命令は再開し、そして完了することができる。したがって、シナリオは次の通りである。データブロックは、あるノードから別のノードに必要に応じて移されるが、ユーザのプロセスからは見えない。つまり、ユーザのプロセスに対して、このシステムは複数のプロセスが自由に読んだり書いたりできる共有メモリを持つ密結合共有メモリマルチプロセッサシステムのように見える。ローカルメモリにデータのコピーがキャッシュされていることによって、キャッシュヒット時、つまりキャッシュに格納されているデータのアドレスをアクセスするときの、メモリアクセスのためのネットワークトラフィックが削減できる。したがって、もしアプリケーションのデータアクセスの局所性が高い割合で生じているときに、ネットワークトラフィックはかなり削減される。

この一般的な方法の変形が、データブロックの複製又は移動、あるいはその両方を DSM システムが認めるかどうかによって異なる様々な実現の中に用い

られている。

5.3 DSMの設計と実装の課題

DSM システムの設計と実装に含まれる重要な問題は以下の通りである。

1. **粒度 (granularity)**:粒度とは DSM システムのブロックサイズのことであり、すなわち共有の単位でありネットワークブロックフォルトが発生したときにネットワークを通して転送されるデータの単位である。可能な単位としては、数ワード、ページ、数ページなどがある。適切なブロックサイズを選択することは、DSM システムにとって重要な部分である。なぜなら、ブロックサイズは並列性の粒度やネットワークブロックフォルトによって生じるネットワークトラフィックの尺度になるからである。

2. **共有メモリ空間の構造 (structure of shared-memory space)**:構造とは、共有データのメモリ内でのレイアウトのことである。DSM システムの共有メモリ空間の構造は、通常 DSM システムがサポートしようとするアプリケーションのタイプに依存する。

3. **メモリ一貫性とアクセス同期 (memory coherence and access synchronization)**:共有データ項目の複製を認める DSM システムにおいては、共有データ項目のコピーが多くのノードのメモリ内で同時に利用される場合がある。この場合、問題となるのは 2 つ以上のノードの主メモリにある共有データの一貫性を扱っているメモリ一貫性問題を解決することである。この問題は、従来のキャッシュでの問題と類似しており [Smith 1982]、特に共有メモリマルチプロセッサにおけるマルチキャッシュとよく似ている [Frank 1984, Goodman 1983, Katz et al. 1985, Yen et al. 1985]。異るメモリ一慣性プロトコルは異る前提とトレードオフを持つため、その選択は通常メモリアクセスのパターンに依存している。また、一慣性 (coherence) と整合性 (consistency) は本書では同じ意味として使う。DSM システムにおいては、共有でーたに対する並行アクセスが起こりうる。そのため、メモリ一慣性プロトコルだけでは、共有データの整合性を維持するのに十分ではない。セナフォ、イベントカウント, そしてロックなどの同期プリミティブを加えることが、共有データの並行アクセスを同期させるためにび必要となる。

4. データ配置とアクセス (data location and access):DSM システムにおいてデータを共有するには、アクセスする利用者によってデータの位置が特定できて取り出せる必要がある。したがって、DSM システムは、使われるメモリ整合性の要件に合うネットワークブロックフォルトをサービスするための、データブロック位置特定機構の何らかの形態での実装が必要となる。

5. 置き換え戦略 (replacement strategy):もし、あるノードのローカルにおけるメモリが満杯のとき、キャッシュミスは遠隔のノードからアクセスされているデータブロックの取り出しのみならず、置き換えを行う必要がある。このとき、ローカルメモリのデータブロックは、新しいデータブロックと置き換えられる。したがって、キャッシュ置き換え戦略も DSM システムの設計には必要となる。

6. スラッシング (thrashing):ある DSM システムにおいては、必要に応じて、データブロックがノード間を移動する。したがって、もし 2 つのノードで 1 つのデータ項目への書き込みアクセスが競合するとき、対応するデータブロックは高い頻度で行き来が起こり、実際の作業が全く進まない場合が生じる。

7. 不均一性 (heterogeneity):同一機種システムに構築された DSM システムは、不均一性を議論する必要はない。しかし、もし基本となるシステム環境が不均一であれば、DSM システムは不均一性に注意して設計されなければならない。そのため、その機能は異なるアーキテクチャを持つマシン上でも正しく動作する。

DSM システムの設計と実装に関わるこれらの問題は次の節に続けて説明する。

5.4　粒度

DSM システムにおいて、選択する際に最も分かりやすいパラメータの 1 つはブロックサイズである。この粒度のパラメータを選択するための幾つかの基準を以下に述べる。単にページ化された主メモリを使う場合、多くのトレードオフがあり、1 つの基準が支配的ということはない。

5.4.1 ブロックサイズの選択に影響する要因

典型的な粗結合マルチプロセッサシステムにおいては、データサイズの大きなパケット (例えば 4 キロバイト) は、小さなパケット (例えば 256 バイト) を送るより非常に高価ということはない。これは、通常典型的なソフトウェアプロトコルと、オペレーティングシステムの仮想メモリ層のオーバヘッドによるものである。このことは、比較的大きなブロックサイズが良いということを意味する。しかし、以下に示すようなブロックサイズの選択に影響する他の要因もある [Nitzberg and Virginia Lo 1991]。

1. **ページングのオーバヘッド**:共有メモリプログラムは参照の局所性を持っているため、1 つのプロセスが大きな共有アドレス空間をアクセスしている時間は非常に短いと考えられる。したがって、大きなブロックサイズのページングのオーバヘッドは小さなブロックサイズのオーバヘッドに比べて小さい。

2. **ディレクトリのサイズ**:ブロックサイズの選択に影響するもう 1 つのファクタは、システムにおいてブロックについてのディレクトリ情報を管理する必要性である。明らかにブロックサイズが大きければ大きいほどディレクトリは小さくなる。このことは、最終的に大きなブロックサイズに対するディレクトリ管理のオーバヘッドは減少することになる。

3. **スラッシング**:スラッシングの問題は、複数のノードによって同時に同じデータブロックにあるデータ項目が更新されるときに発生し、アプリケーションの実行において何の進展もないままに非常に多くのデータブロックの転送が行われる。どのブロックサイズによってもスラッシングの問題は発生するが、大きなブロックサイズの場合、同じブロックの異なる領域が異なるノードのプロセスによって更新されやすくなり、小さなブロックサイズでは必要のないデータブロックの転送が発生する。

4. **偽共有**:偽共有は、2 つの異なるプロセスが同じデータブロックにある 2 つの関連のない変数をアクセスするときに生じる (図 5.2)。このような場合、本来の変数は共有されていないにも拘わらず、データブロックは 2 つのプロセスによって共有されているように見える。ブロックサイズが大きいほど、同じデータブロックに独立に利用される異なるデータ構造が含まれていることで偽

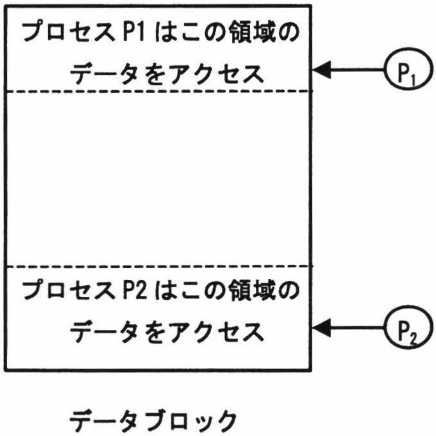


図 **5.2**　偽共有

共有の可能性は高まる。

5.4.2　ブロックサイズとしてのページサイズの利用

DSM 設計者にとって、小さなブロックサイズと大きなブロックサイズとの相対的な利点、欠点は適切なブロックサイズを決定することを難しくしている。そのため、幾つかの既存の DSM システムで用いられている粒度の適切な妥協点は、従来の仮想メモリ実装での典型的なページサイズを DSM システムのブロックサイズとして用いることである。ページサイズを DSM システムのブロックサイズとして用いる場合の利点、欠点は、次のとおりである [Li and Hudak 1989]。

1. DSM ページフォルトにトリガをかけるのに既存のページフォルト手法 (つまりハードウェア機構が使える)。そのため、メモリ一貫性の問題はページフォルトハンドラによって解決されるようになる。

2. アクセス制御機構 (共有されたエンティティ毎に必要となる) は、システムのメモリ管理ユニットの機能に容易に組み込むことができる。

3. 1 つのページが 1 つのパケットに入る限り、ネットワークページフォルト

のときでも、ページサイズによって過度のオーバヘッドにはならない。

4. 実験から、ページサイズはメモリ競合の際の適切なデータエンティティ単位であることが実証されている。

5.5 共有メモリ空間の構造

構造は、DSM システムのアプリケーションプログラマに対して提示する共有メモリ空間の抽象的なビューを定義する。例えば、ある DSM システムの共有メモリ空間はワード単位の記憶領域として見えるかもしれないし、別の DSM システムの共有メモリ空間はデータオブジェクト単位の記憶領域として見えることもある。DSM システムの構造と粒度とは、密接な関係がある [Nitzberg and Virginia Lo 1991]。DSM システムの共有メモリ空間を構造化する 3 つの一般的に用いられている方法は以下の通りである [Nitzberg and Virginia Lo 1991]。

1. **構造化なし**:ほとんどの DSM システムは、共有メモリ空間を構造化していない。これらのシステムでは、共有メモリ空間は単純にワード単位の線形な配列である。構造化されていない共有メモリ空間の利用上の利点は、共有の単位としてどんな適切なページサイズでも選択できる点と、すべてのアプリケーションである固定の粒度が利用できる点である。そのため、このような DSM システムは単純で設計しやすい。また、アプリケーションも必要なデータ構造は何でもこの共有メモリで使うことができる。IVY[Li and Hudak 1989] と Mether[Minnich and Farber 1989] がこの方法をとっている。

2. **データタイプによる構造化**:この方法では、共有メモリ空間はオブジェクトの集まり (Cloud[Dasgupta et al. 1991] そして Orca[Bal et al. 1992] に使われているような)、又はソース言語の変数の集合 (Munin[Bennet et al. 1990] そして Midway[Bershad et al. 1993] にあるような) として構造化される。このような DSM の粒度は、オブジェクト又は変数である。しかし、オブジェクトとデータタイプのサイズは大きく変化するため、これらの DSM システムではアプリケーションによってアクセスされているオブジェクト/変数のサイズに合うように可変の粒度を使う。可変の粒度の使用は、これらの DSM システムの設計と実装を困難にしている。

3. **データベースとしての構造:** もう1つの方法は、共有メモリをデータベースのように構造化する方式である。例えば、Linda[Carriero and Gelernter 1989]はこの方法をとっている。この共有メモリは、**タプルスペース (tuple space)** と呼ばれる連想メモリ (名前やアドレスよりむしろ内容で選択されるメモリ) として整理されている。このタプルスペースとは、型付きのデータ項目をそのフィールドに持つ変化することのないタプルの集まりである。どんなベースとなる言語 (例えば C や FORTRAN) にも追加することのできるプリミティブの集合が提供される。それらは、タプルをタプルスペースを置くときや、それをタプルスペースから読み出したり、取り出したりするのに使う。更新を行うためには、DSM にある古いデータ項目が新しい項目に置き換えられる。プロセスは、タプルのフィールド数や、それらの値や型を指定することで、タプルを選択することができるが、プログラマは共有メモリ空間と相互作用を行うためには特別の関数を使うことが要求される。そのため、共有データへのアクセスは透過的ではない。他のほとんどのシステムは共有データへのアクセスは透過的である。

5.6　一貫性モデル

一貫性の要求は、アプリケーションによって様々である [Bennett et al. 1990, Cheriton 1986, Garcia-Molina and Wiederhold 1982]。**一貫性モデル (consistency model)** とは、基本的にあるアプリケーションの集合に対して正しく動作するメモリのための、共有メモリデータを管理するときの一貫性の度合いをいう。DSM システムが一貫性モデルによって保証された一貫性の度合いを提供するように望むとき、それはアプリケーションが従うべき規則の集合として定義される。既に幾つかの一貫性モデルが提案されている。これらの中から主なものを以下に述べる。

新しい一貫性モデルの開発は、現在の活発な研究領域であることを述べておく。基本的な考えは、既存の一貫性モデルより一貫性の要求がかなりの程度緩和されるような一貫性モデルを開発することである。その緩和は、あるアプリケーションの集合が正しい機能をすることができるという条件で設定される。緩和された一貫性要求によって、並列性が増すために、アプリケーションにとって性能の改善につながる。しかし、もしシステムが弱い一貫性モデルしかサポートしていない場合、より強い一貫性モデルに基づくアプリケーションは正しく動作しない場合がある。これは、もしシステムがより強い一貫性モデ

ルをサポートしているときに、より弱い一貫性モデルは自動的にサポートされるがその逆は真でないことによる。

厳密一貫性モデル

厳密一貫性モデル (strict consistency model) は、最も厳重な一貫性要求を持っており、最も強いメモリ一貫性の形式である。共有メモリシステムは、そのプロセスが読み込みと書き込みを行う場所に関わらず、もしあるメモリアドレスに対する読み込み操作に対して返ってきた値が、常にそのアドレスに対して最後に書き込み操作が行われた値と等しいとき、厳密な一貫性モデルであるという。すなわち、すべての書き込みがすべてのプロセスに同時に見えるようになる。

厳密一貫性モデルの実現には、"最も新しい"という意味を明確にできるように、メモリの読み込み／書き込みの順序を正確に決定できる絶対的なグローバル時間が必要とされる。しかしながら、6章で述べるように、分散システムのすべてのノードのクロックを厳密に同期させることは不可能である。そのため、分散システムにおける絶対グローバル時間も同様に不可能である。結局、DSMシステムに対する厳密一貫性モデルは現実には実現不可能である。

順序一貫性モデル

順序一貫性モデル (sequential consistency model) は、Lamport[1979]によって提案された。共有メモリシステムは、もしすべてのプロセスがすべての共有メモリにおけるメモリアクセス操作を同じ順序で見ていれば、順序一貫性を持つという。メモリアクセス操作がインターリーブされる正確な順序は問題ではない。したがって、もし3つの操作 read(r_1), write(w_1),read(r_2) がある1つのメモリアドレスに対してこの順序で実行されるとき、3つの操作による (r_1, w_1, r_2),(r_1, r_2, w_1),(w_1, r_1, r_2),(w_1, r_2, r_1),(r_2, r_1, w_1),(r_2, w_1, r_1) のどの順番もすべてのプロセスが同じ順序で見るならば起こりうる。もし、1つのプロセスが3つの操作の順序のうちの一つを見て、もう1つのプロセスが異なるプロセスを見ることになれば、そのメモリは順序一貫性のあるメモリではない。ここで注意すべきなのは、厳密一貫性で許されるのは (r_1, w_1, r_2) の順序のみということである。

順序一貫性モデルにおける一貫性の要求は、厳密一貫性のものよりも弱い。それは、順序一貫性のモデルは特定のメモリアドレスに対する読み込み操作が、そのアドレスに対する最も最新の書き込み操作によって書かれた同じ値を

常に返すことを保証していないからである。結局、順序一貫性のメモリを使って、2 つのプログラムを 2 回実行することによって得られる結果は、明示的な同期操作がなければ同じでない可能性がある。この問題は、厳密一貫性メモリでは存在しない。

因果一貫性モデル

因果一貫性モデル (causal consistency model) は、Hutto and Ahamad [1990] によって提案され、順序一貫性モデルの要求を緩めてさらに並列性を高めている。順序一貫性モデルと異なり、因果一貫性モデルでは、すべてのプロセスはメモリ参照操作のみを潜在的に因果関係をある同じ (正しい) 順序で見る。潜在的に因果関係のあるメモリ参照操作は、異なるプロセスには異なる順序で見える可能性がある。あるメモリ参照操作 (読み込み／書き込み) は、もし最初の操作が次の操作に影響を与えるときに、他の参照操作に対して潜在的に因果関係を持つという。例えば、プロセスが読み込み操作を書き込み操作の後に実行するとき、書き込み操作は潜在的に読み込み操作に因果関係を持つ。なぜなら、書き込まれた値の計算が読み込み操作で得られる値に影響するからである。一方、もし最初のプロセスが 2 番目のプロセスによって書かれた値を読み込むことがないか、2 番目のプロセスによって書かれた値から直接あるいは間接的に導出されたどのメモリ変数を読み込むことがないとき、あるプロセスによって実行された書き込み操作は、もう 1 つのプロセスで実行された書き込み操作とは因果関係はない。

共有メモリシステムは、もしすべての潜在的な因果関係のある書き込み操作がすべてのプロセスにおいて同じ (正しい) 順序で見えるとき、因果一貫性モデルをサポートするという。潜在的な因果関係のない書き込み操作は異なるプロセスには異なる順序に見える可能性がある。" 正しい順序 " の意味は、もし書き込み操作 (w_2) がもう 1 つの書き込み操作 (w_1) に因果関係があるとき、可能な順序は (w_1, w_2) である。なぜなら、w_2 によって書き込まれた値が w_1 によって書かれた値によっていつか影響を受ける可能性があるからである。そのため、(w_2, w_1) は認められない順序である。

明らかに、因果一貫性をサポートする共有メモリシステムの実装において、メモリ参照操作がどの他のメモリ参照操作に依存しているかを追跡することが必要となる。これは、メモリアクセス操作の依存関係グラフを構成して管理することで可能となる。

パイプラインランダムアクセスメモリ一貫性モデル

パイプラインランダムアクセスメモリ (**PRAM**) 一貫性モデル (**pipelined random-access memory (PRAM) consistency model**) は、Lipton and Sandberg[1988] によって提案され、これまで述べてきた一貫性モデルより弱い一貫性セマンティクスを提供する。このモデルは、1 つのプロセスで実行されるすべての書き込み操作が、すべての他のプロセスからはその操作が 1 つのパイプラインにあるように実行される順序で見えることが保証されるだけである。異なるプロセスで実行される書き込み操作は、異なるプロセスからは異なる順序で見える。例えば、もし w_{11} と w_{12} がプロセス P_1 によってこの順番で実行される 2 つの書き込み操作とする。そして、w_{21} と w_{22} はプロセス P_2 によってこの順序で実行される 2 つの書き込み操作とする。プロセス P_3 は、それらを $[(w_{11}, w_{12}), (w_{21}, w_{22})]$ の順序で見るかもしれないし、プロセス P_4 は $[(w_{21}, w_{22}), (w_{11}, w_{12})]$ の順序で見るかもしれない。

PRAM 一貫性モデルは、簡単で実装が容易であり、そして良い性能を持っている。このモデルは、各ノードで他のノードと独立に実行された書き込み操作を簡単に順番に並べるだけで実現することができる。このモデルは以前のモデルより良い性能を示す。それは、1 つのプロセスによるすべての書き込み操作がパイプライン化されるので、次の操作を開始する前に書き込み操作を完了するまで待つ必要がないためである。順序一貫性においては、すべてのプロセスがメモリ参照の操作の順番を合意しているが、PRAM 一貫性ではすべてのプロセスがメモリ参照操作の順番を合意してない。したがって、上で示した例に対して、$[(w_{11}, w_{12}), (w_{21}, w_{22})]$ 又は $[(w_{21}, w_{22}), (w_{11}, w_{12})]$ は順序一貫性では可能な順番であるが、両方が可能ではない。このことは、PRAM 一貫性とは異なり P_3 と P_4 は同じ順序を合意している必要がある。

プロセッサ一貫性モデル

プロセッサ一貫性モデル [processor consistency model] は、Goodman[1989] により提案され、PRAM 一貫性にメモリ一貫性の付加的な制約を加えたものに近い。これは、プロセッサ一貫性を持つメモリは、整合性もあり PRAM 一貫性モデルも持っている。メモリ整合性は、どんなメモリ位置に対しても、すべてのプロセスはその位置におけるすべての書き込み操作の順番が同じであることの合意を持っているということである。実際、プロセッサ一貫性は同じメモリ位置におけるすべての書き込み操作は (それをどのプロセスが実行したかによらず) すべてのプロセスに同じ順番で見えることを保証している。この要求は、

PRAM 一貫性モデルによって示される要求に加えられる。したがって、PRAM 一貫性で使った例において、もし w_{12} と w_{22} が同じメモリ位置 x に対する書き込み操作とすると、すべてのプロセスはそれらを同じ順番で見えなければならない。つまり、w_{12} が w_{22} の前か、w_{22} が w_{12} の前かである。これは、このことは、プロセッサ一貫性では、P_3 と P_4 は書き込み操作は同じ順番で見なければいけない。これは、$[(w_{11}, w_{12}), (w_{21}, w_{22})]$ 又は $[(w_{21}, w_{22}), (w_{11}, w_{12})]$ のどちらかとなる。この例では、プロセッサ一貫性と順序一貫性は同じ結果をもたらしているが、他の場合では異なることもある。

弱一貫性モデル

弱一貫性モデル (weak consistency model) は、Dubois et al[1988] によって提案され、多くのアプリケーションに共通の以下の 2 つの特性を活用するように設計されている。

1. 他のプロセスに各書き込み操作によってなされた変更を見せる必要はない。幾つかの書き込み操作は、省略することができる。必要になった時点で、他のプロセスに送ることができる。例えば、プロセスが臨界領域を実行しているとき、そのプロセスが臨界領域を出るまで他のプロセスはメモリに対する変更を見ることは仮定されない。この場合、プロセスによってメモリに行われたすべての変更は、プロセスが臨界領域を出るときにのみ見えるようになる。

2. 共有変数に対する独立なアクセスはまれである。これは、多くのアプリケーションでは、プロセスは共有変数の集合に対して複数のアクセスを行い、次に長い時間、この集合の変数にアクセスを全く行わないということである。

両方の特性は、独立したメモリ参照操作よりは、グループ化されたメモリ参照を行わせることで、よりよい性能を達成できることを意味している。

この考えを実現する上での主な問題は、プロセスによって実行された変更を他のプロセスに見せる時間をどのようにシステムが知るかを決定することである。なぜなら、この時間はアプリケーションによって異なるからである。プログラマは、自分のアプリケーションにおけるこの時間をシステムに知らせることが求められている。これに対して、弱一貫性モデルをサポートする DSM システムは、**同期変数 (synchronization variable)** という特別な変数を用いる。この変数の操作が、メモリ同期に使われる。これは、同期変数がプロセス

によってアクセスされると、全(共有)メモリが全プロセスによってなされた全変更が他の全プロセスに見えるようにすることで、同期がとられる。DSMシステムにおけるメモリ同期には、あるノードでなされたメモリ更新を同じメモリアドレスのコピーを持つすべての他のノードへ送付することも含む。

弱一貫性をサポートするには、以下の要件が満たされる必要がある。

1. すべての同期変数へのアクセスは、順序一貫性セマンティクスに従う必要がある。
2. すべての以前の書き込み操作は、どの場所での同期変数へのアクセスが許される前に完了している必要がある。
3. すべての以前の同期変数へのアクセスは、同期変数でない変数へのアクセスが許される前に完了している必要がある。

弱一貫性モデルは、プログラマに追加の努力を強いるコストがかかる分、良い性能を提供する。

リリース一貫性

弱一貫性モデルでは、全(共有)メモリがプロセスによって同期変数をアクセスされたときに同期がとられ、メモリ同期には以下の操作が含まれることを見てきた。

1. プロセスによって加わったすべての変更は、他のノードに伝搬する。
2. 他のプロセスによってメモリに加わったすべての変更は、他のノードからそのプロセスのノードに伝搬する。

最近の研究では、これらのことが必ずしも必要ではないことが分かっている。その理由は、最初の操作はプロセスが臨界領域から出るときにのみ必要であり、2番目の操作は臨界領域に入るときのみ必要だからである。弱一貫性モデルでは、唯一の同期変数が使われているため、同期変数をアクセスしているプロセスが臨界領域から出ようとしているのか、入ろうとしているのかがシステムには分からない。したがって、最初と2番目の操作の両方を、プロセスによって同期変数がアクセスされたときに実行する。性能を向上させるために、**リリース一貫性モデル (release consistency model)**[Gharachorloo et al. 1990]は、プロセスが臨界領域から出るのか入るのかをシステムに明確に伝え、それによりプロセスが同期変数をアクセスしたときに最初の操作か2番目の操作かを

決定しそれを実行することができる。このことは、2 つの同期変数 (acquire と release と呼ばれる) を従来の 1 つの同期変数の代わりに使うことで実現される。acquire はプロセスが臨界領域に入ることをシステム伝えるために使われ、この変数がアクセスされたときは 2 番目の操作だけを実行する。一方で、release はプロセスが臨界領域を出たことをシステムに伝えるために使われ、この変数がアクセスされたときは最初の操作だけが実行される。プログラマは、自分のプログラムの適切な場所に acquire と release を入れることに責任を持つ。

リリース一貫性は、臨界領域に代わってバリアに基づく同期機構を使っても実現される。**バリヤ (barrier)** とは、並行プロセスのグループ実行フェーズの終わりを定義する。グループ中のすべてのプロセスは、バリヤに続く実行をどれかのプロセスが許される前に、その実行をバリヤまで完了しなければならない。つまり、グループのプロセスがその実行中にバリアに出会ったならば、すべてのグループの他のプロセスがバリヤまで実行を完了するまでブロックする。最後のプロセスがバリヤまで実行を完了したとき、すべての共有変数は同期がとれ、すべてのプロセスは実行を再開する。したがって、acquire はバリヤからの脱出、release は最後のプロセスのバリヤまでの実行の完了となる。

バリヤは、集中型のバリヤサーバを使って実現することができる。バリヤが作られると、開放されるまでバリヤを待たなければならないプロセス数のカウンタが得られる。並行プロセスのグループの各プロセスは、バリヤに出会うと、バリヤサーバにメッセージを送信し、バリヤサーバからの応答を受信するまでブロックする。バリヤサーバは、グループ内のすべてのプロセスがバリヤに到達しなければ、応答は返さない。

リリース一貫性をサポートするためには、以下の要件が満たされる必要がある。

1. すべての acquire 及び release 同期変数へのアクセスはプロセッサ一貫性セマンティクスに従う。
2. プロセスによって以前に実行された acquire は、メモリ上のデータアクセス操作の実行が認められる前に正常に完了している必要がある。
3. プロセスによるデータアクセス操作の以前の実行は、プロセスによって release アクセスの実行が認められる前に正常に完了している必要がある。

異なる臨界領域 (又はバリヤ) のロックについての acquire や release へのアクセスは、互いに独立に発生することができる。Gharachorloo et al.[1990] は、

もし適切な同期アクセスをきちんと用いていれば、リリース一貫性によるDSMシステムは、順序一貫性のDSMシステム上でのアプリケーションを実行させたときと同じ結果となる。

リリース一貫性の変形として、**遅延リリース一貫性 (lazy release consistency)** が Keleher et al.[1992] によって提案されており、それは通常のリリース一貫性よりもより効率的になっている。既存の方法だと、プロセスがreleaseアクセスを行ったとき、そのプロセスのノードにおいてすべての変更されたメモリの位置の内容が、同じメモリ位置をローカルキャッシュにコピーとして持っている他のすべてのノードに送られる。しかし、遅延方式では、変更はリリースの時点では他のノードに送られない。その代わり、これらの変更は必要に応じて他のノードに送られる。これは、プロセスがacquireアクセスを行ったときに、すべての他のノードによる変更がそのプロセスのノードに獲得されるというものである。したがって、releaseアクセスの時点で、このノードに送られるべき変更は、そのノードに今 (正確にいうと必要になったとき) 受信される。遅延リリース一貫性は、他のどれかのノードでのプロセスによってacquireアクセスが行われるまでネットワークトラフィックが発生しないことから、よりよい性能を持っている。

一貫性モデルの議論

今までの記述によって、主な一貫性モデルの幾つかを見てきた。他にも幾つかの方法が提案されてきている。一貫性モデルの良い概要としては、[Mosberger 1993] がある。一貫性モデルを性能によって順位付けするのは、異なるアプリケーションに対して通常は非常に異なる結果が得られるために困難である。つまり、あるアプリケーションはあるモデルで非常に良い性能が得られるが、他のアプリケーションは他のモデルで良い性能を持つことがある。したがって、DSMシステムの設計においては、実装の容易さ、そのセマンティクスの直感との近さ、どれくらいの並列性を許すか、そして使いやすさ (プログラマに追加の努力を求めるものか) などの幾つかの他の要素に依存して一貫性モデルの選択が行われる。

上で述べた一貫性モデルの中では、厳密一貫性はその実現が現実的には困難であるため、DSMシステムの設計には決して使われない。DSMシステムに最もよく使われるモデルは順序一貫性モデルである。それは、実現可能であり、メモリ整合性に対して最も直感的に期待されるセマンティクスをサポートしているからであり、そしてプログラマに追加の努力を一切要求していないからで

ある。その人気のもう 1 つの重要な理由は、順序一貫性を持つ DSM システムは、既存のマルチプロセッサプログラムを変更無しにマルチコンピュータ上で動作させることができる点である。ここでは、マルチプロセッサ向けに書かれたプログラムは、通常メモリが順序一貫性を持っていることを仮定しているからである。しかし、この方式は非常に制限があり、したがって低い並行性という欠点がある。したがって、幾つかの DSM システムは順序一貫性より弱い他の一貫性モデルを使うように設計されている。

因果一貫性、PRAM 一貫性、プロセッサ一貫性、弱一貫性、そしてリリース一貫性はより弱い範疇の中の主なものである。DSM システムを設計するときに因果一貫性、PRAM 一貫性、そしてプロセッサ一貫性を使う上での問題点は、異なるプロセスが異なる操作の順序を見るために、メモリの整合性に関する直感的に期待されるセマンティクスをサポートしていないことである。したがって、これらの 3 つのモデルを使うには、プログラマの責任で順序一貫性のメモリのみで動作することを避けるようにしなければならない。これについては、これらのモデルはプログラマに追加の労力を要請することになる。

弱一貫性とリリース一貫性モデルは、明示的な同期変数を用いており、よりよい並行性を提供し、直感的に期待されるセマンティクスをサポートしていることから、DSM 設計に用いることについてより可能性がある。これらの一貫性モデルの問題は、プログラマが同期変数を適切に使わなければならない点である。これは、プログラマの負担となる。

DSM システムを実装するための困難に立ち向う主な理由は、共有メモリパラダイムが分散アプリケーションのプログラミングをメッセージパッシングパラダイムよりも簡単にすることである。したがって、幾つかの DSM システムの設計者は、プログラマには追加の労力は負担させるべきではないという意見を持つ人々である。この観点を持つ設計者は、順序一貫性モデルの利用を好む。

5.6.1　順序一貫性モデルの実装

ほとんど一般に用いられている DSM システムの一貫性モデルは順序一貫性モデルであることを見てきた。そのため、順序一貫性を持つ DSM システムを実現するために一般に使われているプロトコルの記述は、以下に説明する。リリース一貫性を持つ DSM システムの実装のためのプロトコルは、次の項で述べる。

順序一貫性モデルを DSM システムに実現するためのプロトコルは、大部分

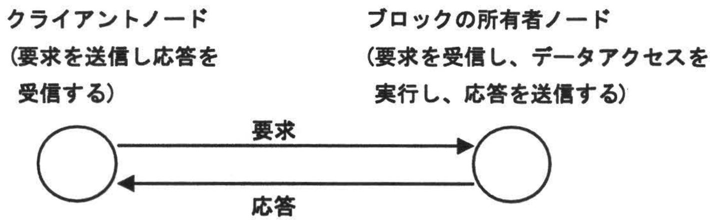


図 **5.3**　複製なしで、移動なしのブロック (NRNMB) 戦略

はDSMシステムが共有メモリデータブロックの複製そして移動を認めるかどうかに依存している。DSMシステムの設計者は、以下の複製及び移動の戦略の間で選択を行っている [Stumm and Zhou 1990]。

1. **複製なしで、移動なしブロック (NRNMB)**
2. **複製なしで、移動ありブロック (NRMB)**
3. **複製ありで、移動ありブロック (RMB)**
4. **複製ありで、移動なしブロック (RNMB)**

これらの各カテゴリに使われるプロトコルを以下に述べる。各カテゴリに適したデータ位置決め機構についても述べる。以下に説明する考えは、IVYシステム [Li 1988] によって最初に提案され、使われた。

複製なしで、移動なしブロック

これは、順序一貫性を持つDSMシステムを実装するための最も単純な方法である。この戦略では、共有メモリの各ブロックは唯一のコピーを持ち、その位置は通常固定されている。すべてのノードからのブロックへのアクセス要求は、ブロックの所有者ノード **(owner node)** に送信される。そのノードは、そのブロックの唯一のコピーを持っているノードである。クライアントノードからの要求を受信すると、所有者ノードのメモリ管理ユニット (MMU) とオペレーティングシステムソフトウェアは、ブロックへのアクセス要求を実行して、クライアントに応答を返す (図 5.3)。

共有ブロックを持つノードが、すべての (ブロックへの) アクセス要求を受信した順序で実行するため、順序一貫性が維持されることは明白である。

この方法は簡単で、実装するのも容易であるが、以下の欠点がある。

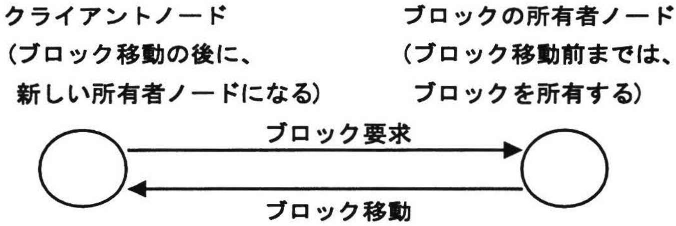


図 **5.4**　複製なしで、移動ブロック (NRMB) 戦略

■データアクセスの直列化がボトルネックになる
■ DSM の主な利点である並列性がこの方法では不可能である。

NRNMB 戦略におけるデータ位置決め: NRMB 戦略は、以下のような特徴を持っている。

1. 全体のシステム内に、各ブロックは 1 つのコピーしかない。
2. ブロックの位置は決して変化しない。

これらの特徴に基づいて、この場合の最良のブロックの位置決め方法は、ブロックからノードへの簡単なマッピング機能を使うことである。フォルトが発生したとき、フォルトが発生したノードのフォルトハンドラは、アクセスするブロックの位置を得るためにマッピング機能を使い、アクセス要求をそのノードに送る。

複製なしで、移動ありブロック

この戦略では、共有メモリの各ブロックはシステム全体で 1 つのコピーを持つ。しかし、ブロックへの各アクセスは現在のノードから要求があったノードへの移動を生じる。したがって、ブロックの所有者ノードが固定されたままの前の戦略とは異なり、この戦略ではブロックの所有者ノードは、ブロックが新しいノードに移動するとすぐ変化する (図 5.4)。ブロックが移動した場合、それがマップされていたすべてのローカルアドレス空間が削除される。この戦略では、1 つのノードで実行しているプロセスのみが読み込み、又は書き込みをデータ項目に対していつでも実行できる。したがって、この方法は順序一貫性を保証している。

この方法は、以下の利点を持っている [Stumm and Zhou 1990]。

1. 1つのプロセスが現在ローカルに持っているデータをアクセスするのには通信コストはかからない。
2. アプリケーションのデータアクセスの局所性の利点を活かすことができる。もし、アプリケーションが高い参照の局所性を示すとき、データ移動のコストは複数のアクセスによって相殺される。

しかし、この方法は以下の欠点を持っている。

1. スラッシング問題を孕んでいる。つまり、あるノードから別のノードに頻繁に移動を行い、結果的に移動の間のほんの少しのメモリアクセスしか行われないで、性能が劣化する。
2. この方法においても、並列性の利点は活かされない。

NRMB 戦略におけるデータ位置決め: NRMB 戦略においては、各ブロックには単一のコピーしかないが、ブロックの位置は動的に変化する。そのため、以下の方法のうちの一つがデータを位置決めするのに使われる。

1. **ブロードキャスト (broadcasting)**:この方法では、各ノードは**所有ブロックテーブル (owned blocks table)** を管理し、そこの各ブロックの欄には現在の所有者が記載される (図 5.5)。フォルトが発生すると、フォルト発生したノードのフォルトハンドラがブロードキャストの読み込み／書き込み要求をネットワークに送信する。要求されたブロックの所有者は、要求しているノードに対してブロックを送信することでブロードキャスト要求に応答する。
ブロードキャストアルゴリズムの主な欠点は、規模拡張性が良くないことである。要求がブロードキャストされると、要求されたブロックを持っていないすべてのノードがブロードキャスト要求を処理しなければならない。これは、通信サブシステムが潜在的なボトルネックになる。ブロードキャストのネットワーク遅延も完了するまでに長い時間がかかる。

2. **集中サーバアルゴリズム (centralized-server algorithm)**:この方法では、集中サーバが、共有メモリ空間にあるすべてのブロックの位置情報を含むブロックテーブルを管理する (図 5.6)。集中サーバの位置と識別子はすべての

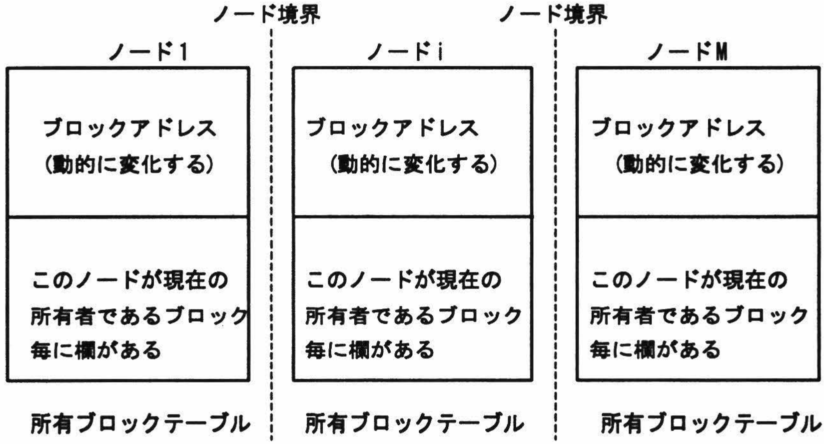


図 5.5 NRMB 戦略のための放送型データ位置位置決め機構における所有ブロックテーブルの構造と位置

ノードによく知られているものとする。

フォルトが発生すると、フォルトを発生したノード (N) のフォルトハンドラは、アクセスしたブロックの要求を集中サーバに送る。集中サーバは要求されたブロックの位置情報をブロックテーブルから取り出して、そのノードに要求を転送する。そして、ブロックテーブルの対応する欄の位置情報をノード N に変更する。要求を受信すると、現在の所有者はブロックをノード N に送り、そこが新しいブロックの所有者になる。

集中サーバ方式は、以下の 2 つの欠点がある。

(a) 集中サーバは位置問い合わせを直列化し、並列性を減らしている。
(b) 集中サーバの障害によって、DSM システムは機能を停止する。

3. 固定分散サーバアルゴリズム (fixed distributed-server algorithm): この方式は、集中サーバ方式の直接の拡張である。この方式は、集中サーバの役割を、幾つかの分散サーバに分散することで、集中サーバ方式の欠点を克服している。したがって、この方法では幾つかのノードにブロック管理者が存在し、各ブロック管理者は、予め決められているブロックのサブセットを管理する (図 5.7)。ブロックと特定のブロック管理者及びそのノードを対応づける機能をマッピング機能という。フォルトが発生すると、フォルトハンドラによっ

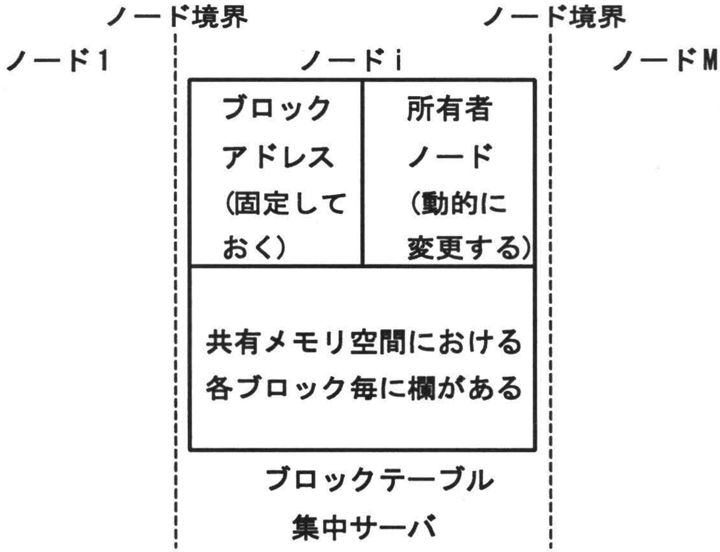


図 5.6　NRMB 戦略のための集中型サーバによるデータ位置位置決め機構におけるブロックテーブルの構造と位置

てマッピング機能が現在の要求しているブロックを管理しているブロック管理者を見つけるのに使われる。そして、アクセスしているブロックへの要求は、そのノードのブロック管理者に送られる。ブロック管理者は、要求を集中サーバアルゴリズムと同じ方法で扱う。

4. 動的分散サーバアルゴリズム (dynamic distributed-server algorithm):この方式は、ブロック管理者を利用せず、各ノードにおいてすべてのブロックの所有者情報を追跡する。このために、共有メモリ空間のすべてのブロックに対する欄を持つブロックテーブルを各ノードは持っている (図 5.8)。しかし、所有者フィールドの情報はいつでも正確である必要はない。しかし、もし正しくない場合は、少なくともブロックの正しい所有者ノードにたどっていけるためのノードの列の最初ノードを提供している。したがって、このフィールドはブロックの所有者の位置のヒントを与えており、したがって**予想所有者 (probable owner)** と呼ぶ。

フォルトが発生したとき、フォルトを発生したノード (N) のフォルトハンドラは、ローカルブロックテーブルのアクセスされたブロックに対する予想所有者ノード情報を取り出し、そのノードにブロック要求を送る。もし、そのノー

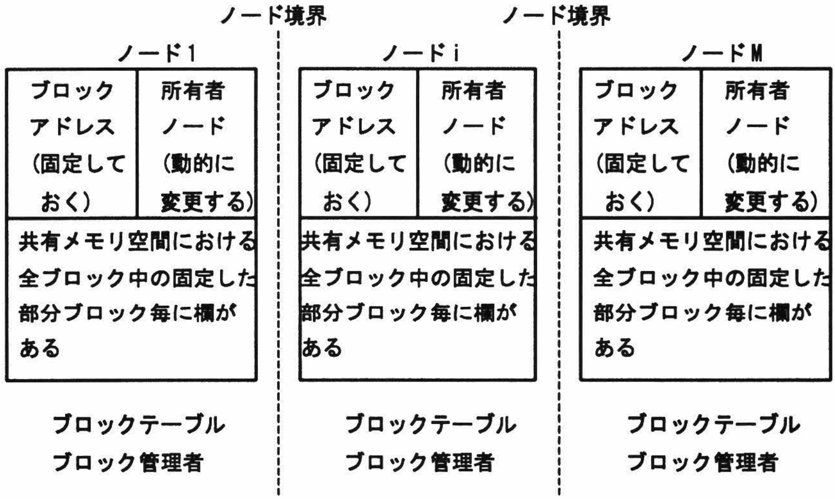


図 5.7　NRMB 戦略のための固定分散サーバによるデータ位置位置決め機構におけるブロックテーブルの構造と位置

ドがそのブロックの正しい所有者であるとき、そのノードはブロックをノードNにおくり、ローカルのブロックテーブルにおけるそのブロックの位置情報をノードNにする。そうでないとき、そのノードはローカルのブロックテーブルを引いて、そのブロックの欄の予想所有者フィールドに示されているノードに要求を転送する。そして、このフィールドをノードNに更新する。ノードNがブロックを受け取ったとき、ノードNは新しいブロックの所有者になる。

複製ありで、移動ありブロック

複製なし戦略の大きな欠点は、ブロックを含むデータにアクセスできるのはいつでも1つのノード上のプロセスであることから、並列性が欠如していることである。並列性を増加させるために、すべての DSM システムはブロックを複製している。複製されたブロックについては、読み込み操作はデータのアクセスにローカルのコピーを使うことによって、複数のノードによる並列処理が行われる。したがって、データの複製がローカルノードにある場合、通信オーバヘッドがなくなるため、読み込み操作の平均コストは削減される。しかし、あるブロックに対する書き込みは、一貫性のためにすべての複製を無効化するか更新するかする必要があるため、書き込み操作のコストは増加する傾向にあ

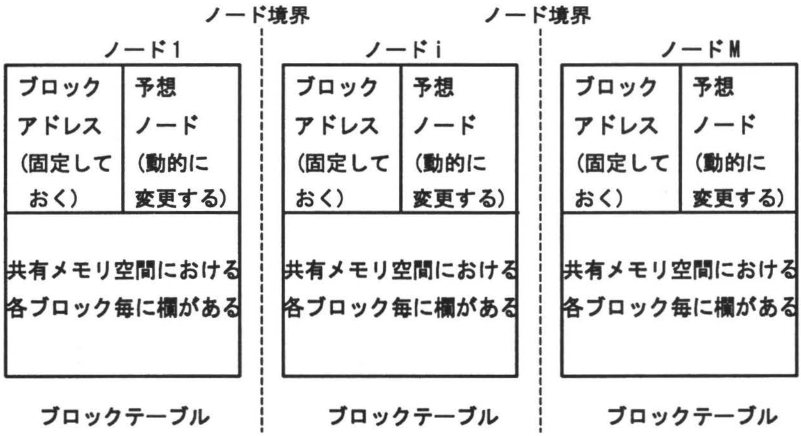


図 5.8 NRMB 戦略のための動的分散サーバによるデータ位置位置決め機構におけるブロックテーブルの構造と位置

る。それにも拘わらず、読み込み対書き込みの比率が大きい場合、書き込み操作に対する余分の手間は読み込み操作の小さい平均コストによって相殺される。

複製は、ブロックの複数のコピーの一貫性を維持する要求のために、メモリ整合性プロトコルを複雑にしている。以下のように、2つの基本的なプロトコルが順序一貫性を保証するのに使われている。

1. **書き込み無効化 (write-invalidate)**:この方法では、すべてのデータのコピーは1つのノードを除いて書き込み前に無効化される。したがって、あるノードで書き込みフォルトが発生したとき、そのフォルトハンドラがそのブロックの現在の所有者ノードからアクセスしているブロックをコピーし、そのブロックを持っているすべての他のノードに対してブロックアドレスを含む無効化メッセージを送信してそのコピーを無効化し、そのブロックのローカルコピーに対して書き込みアクセスをし、そしてフォルトが発生した命令に戻る(図 5.9)。戻った後で、そのノードはそのブロックを“所有”し、その書き込み操作そして他の読み込み／書き込み操作をブロックの所有権が他のノードに放棄されるまで進めることができる。この方法においては、ブロックの無効化の後、そのブロックに書き込み操作を実施するノードのみがそのブロックの変更版を持っていることが重要である。もし、無効化前にそのブロックのコピーを持っているノードの1つが無効化の後にメモリアクセス (読み込みや書き込み)

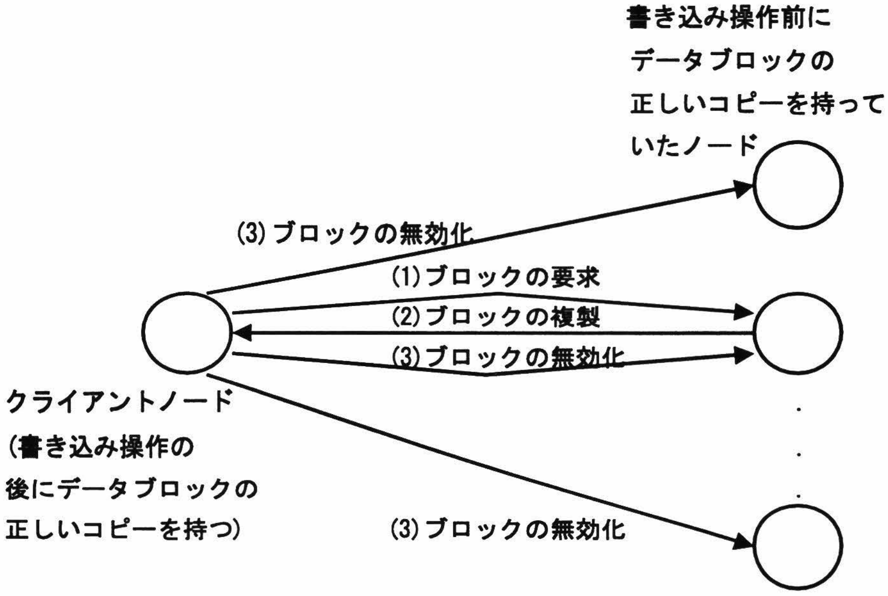


図 5.9　複製移動ブロック (RMB) 戦略のための書き込み-無効化によるメモリ整合性方式

を実行しようとすると、キャッシュミスが起こりそのノードのフォルトハンドラがそのブロックの正しいコピーを持っているノードからそのブロックを取ってくることになる。したがって、この方法は順序一貫性を達成している。

2. **書き込み更新 (write-update)**:この方式では、書き込み操作は書き込みが行われるデータのすべてのコピーが更新される。したがって、1 つのノードで書き込みフォルトが発生すると、フォルトハンドラはブロックが現在あるノードのうちのどれかのノードから自分のノードにアクセスされたブロックをコピーする。ブロックのローカルコピーに書き込み操作を行って、またそのブロックを持っているノードすべてに変更されたメモリ位置のアドレスと新しい値を送信することで、すべてのコピーを更新する。そして、フォルトした命令に戻る (図 5.10)。書き込み操作は、すべてのブロックのコピーが成功裏に更新されたときに完了する。この方法では、書き込み操作が完了した後では、書き込み前にブロックのコピーを持っていたノードは書き込み後でも正しい書き込み後の値を持つ。この方法では、すべてのノードの書き込みが線順序付けされ

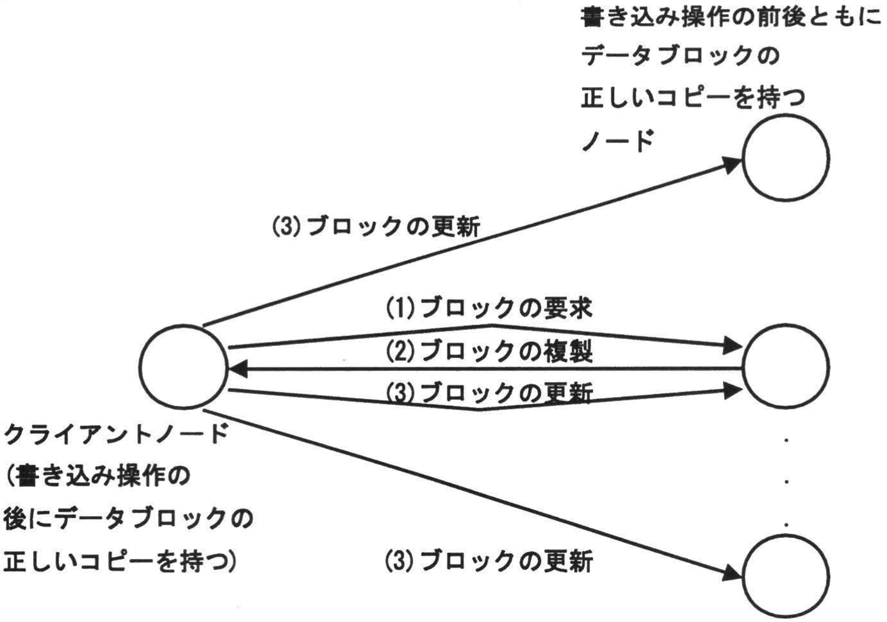


図 5.10　複製移動ブロック (RMB) 戦略のための書き込み-更新によるメモリ整合性方式

る機構を使うことで、順序一貫性を達成しているため、すべてのアクセスは書き込み順序を合意している。これを行う方法の 1 つは、グローバルシーケンサをすべてのノードへの書き込みの順序付けに使うことである。この方式では、各書き込み操作での変更は、最初にグローバルシーケンサに送られる。シーケンサは、変更に順序番号を与え、そして変更にこの順序番号を付けて変更されるべきデータの複製を持つすべてのノードに送る (図 5.11)。書き込み操作は、各ノードで順序番号どおり実行される。ここで、新しい変更がノードに到達したときは、その順序番号が次に予期されたものかどうかを検査する。もし、その検査が失敗したときは、変更が失われているか、変更の順番が間違っているかであり、その場合シーケンサに失った変更の再送を要求する。明らかに最近の書き込み操作のログがどこかに保持されている必要がある。どの 2 つの連続する書き込み操作の間にある読み込み操作は、自明であり、順序一貫性に対するそれらの順序付けは容易に行える。

書き込み更新方式は、書き込み操作にネットワークアクセスを必要とし、変

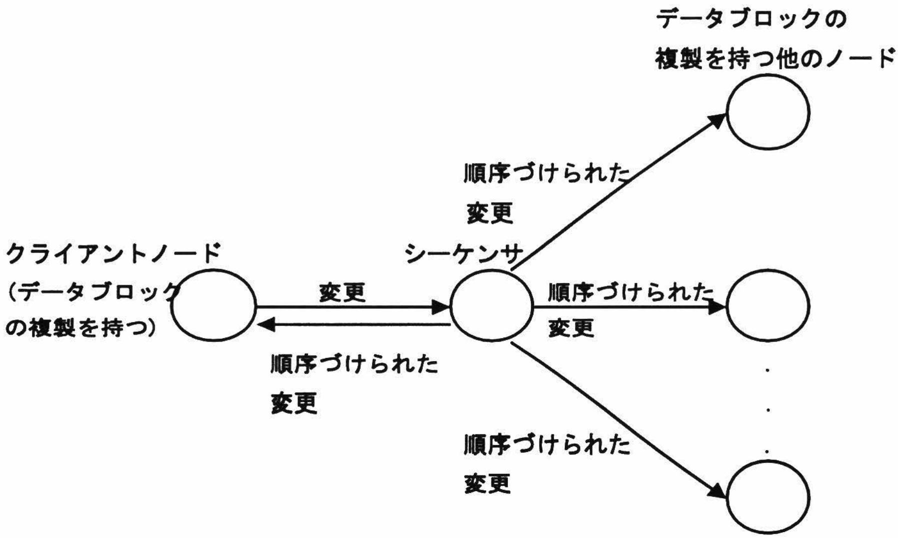


図 5.11　すべてのノードにおいて書き込み操作を順序づけるためのグローバルシーケンサ機構

更されたブロックのすべてのコピーの更新を必要とするために、疎結合分散メモリシステムにとって非常に負担がかかる。他方、書き込み無効化方式では、更新はデータが読み込まれるときにのみ伝搬し、通信が必要になる前に幾つかの更新が起こることもある (なぜなら、多くのプログラムは参照の局所性を持つので)。したがって、ほとんどの DSM システムは、書き込み無効化プロトコルを使っている。書き込み無効化プロトコルの基本的な実装方法では、書くブロックに対応する状態タグを使う。状態タグは、ブロックが正しいかどうか、共有されているかどうか、読み込み専用か、書き込み可能かを示す。この状態情報を使い、読み込みと書き込みが以下の手順で実行される [Nitzberg and Virginia Lo 1991]。

読み込み要求

1. もしデータを含むローカルブロックがあり、もしそれが正しいなら、データのローカルコピーをアクセスして要求を満たす。

2. さもなければ、要求ノードのフォルトハンドラが読み込みフォルトが発生し、そしてそのブロックの正しいコピーを持つノードからブロックを入手する。

もし、そのブロックが他のノードから書き込み可能なら、この読み込み要求は読み込み専用とする。読み込み要求は、ローカルブロックのデータをアクセスすることで満たされる。これは、無効化要求がくるまで正しい状態とする。

書き込み要求

1. もしデータを含むブロックがあり、それが正しくて書き込み可能なら、その要求はデータのローカルコピーをアクセスすることですぐに満たす。

2. さもなければ、要求しているノードのフォルトハンドラが書き込みフォルトを発生し、ブロックの正しいコピーを入手して、その状態を書き込み可能とする。そのブロックの書き込みフォルトは、そのブロック以外のすべてのコピーの無効化を行う。ブロック以外のすべてのコピーの無効化が完了した時点で、そのブロックはローカルで正しく、そして書き込み可能となる。そして、初めの書き込み要求が実行される。

RMB 戦略におけるデータ位置決め: RMB 戦略に使われる書き込み無効化プロトコルは、以下のデータ位置決め問題を含む。

1. ブロックの所有者の位置の特定。ブロックの**所有者 (owner)** とは、ブロックを所有し、言わば最も最後にそれに書き込みアクセスを行ったノードである。

2. ブロックの正しいコピーを持っているノードの追跡。

これら 2 つの問題を解決するために、以下のアルゴリズムの 1 つが使われる。

1. **ブロードキャスト (broadcasting)**:この方法では、各ノードはブロックテーブル (図 5.12) を持っている。ノードのテーブルは各ブロックに対する所有者が示されている欄がある。このテーブルの各欄は、**コピー集合 (copy set)** フィールドがあり、対応するブロックの正しいコピーを持っているノードのリストが含まれている。

読み込みフォルトが発生すると、フォルトが発生したノード (N) が必要なブロックの所有者を見つけるためにネットワーク上にブロードキャストの読み込み要求を送信する。ブロックの所有者は、自身のブロックテーブルにあるそのブロックのコピー集合フィールドにノードを追加し、ノード N に対してコピーを送信する。同様に、書き込みフォルトが発生した場合、フォルトしたノードはブロードキャストの書き込み要求を必要なブロックについて送信する。この

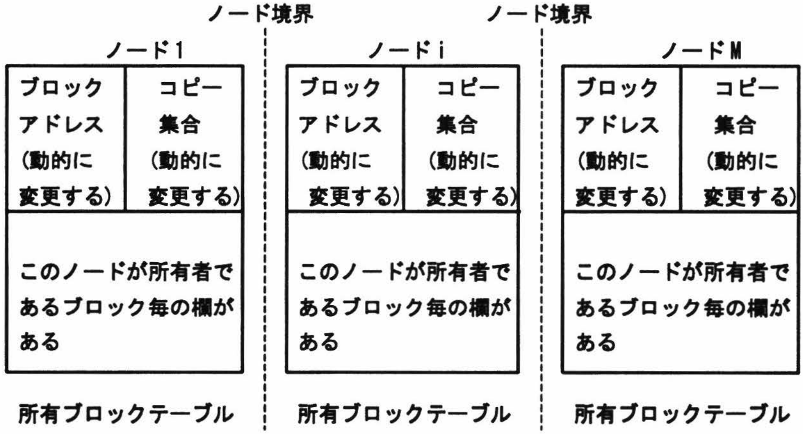


図 5.12 RMB 戦略のための放送型データ位置特定機構における所有ブロックテーブルの構造と位置

要求を受信したとき、ブロックの所有者はこの所有権をノード N に放棄して、そのブロックとコピー集合をノード N に送る。そのブロックとそのコピー集合をノード N が受信したとき、ノード N はコピー集合にあるすべてのノードに無効化メッセージを送信する。これで、ノード N はこのブロックの新しい所有者となり、したがってローカルに持っているブロックテーブルに 1 つの欄が追加される。この欄のコピー集合フィールドは、すべてのコピーが無効化されているので、ブロックの他のコピーはないことを示すように初期化される。

このブロードキャストの方法は、NRMB 戦略について、データの位置決め機構の説明で述べている欠点を抱える方法である。

2. **集中サーバアルゴリズム (centralized-server algorithm)**。この方法は、NRMB 戦略の集中サーバアルゴリズムに似ている。しかし、この場合のブロックテーブルの各欄は、集中サーバによって管理され、ブロックの現在の所有者を示す**所有者ノード (owner node)** フィールド、ブロックの正しいコピーを持っているノードのリストを含むコピー集合フィールドを持つ (図 5.13)。

読み込み／書き込みフォルトが発生すると、フォルトを発生したノードは、アクセスしたブロックの読み込み／書き込みフォルト要求を集中サーバに送る。読み込みフォルトに対して、集中サーバはノード N をそのブロックのコピー集合に加え、ノード N に所有者ノード情報を返す。他方、書き込みフォルトにつ

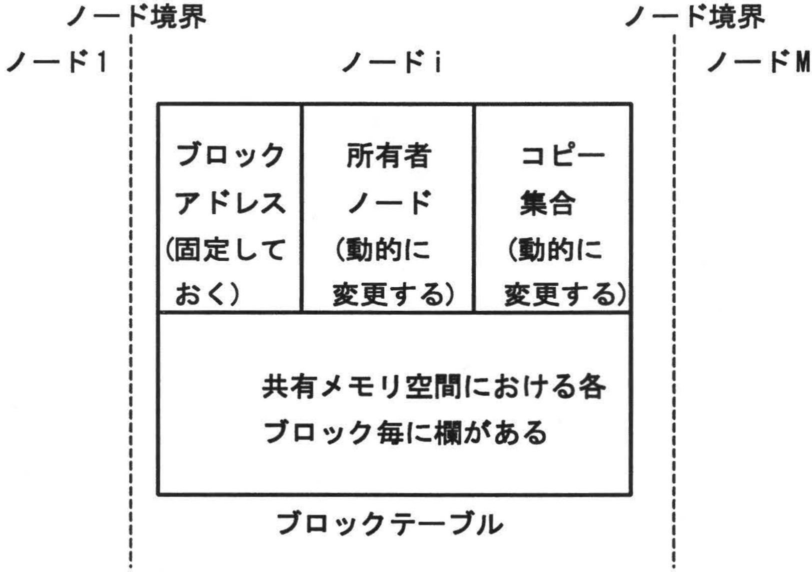


図 5.13 RMB 戦略のための集中サーバによるデータ位置位置決め機構におけるブロックテーブルの構造と位置

いては、コピー集合と所有者ノード情報をノード N に返し、そしてコピー集合フィールドをノード N のみを含むように初期化する。ノード N は、次に所有者ノードにブロックの要求を送る。この要求を受けて、所有者ノードはノード N にブロックのコピーを送る。ノード N は、その後読み込み／書き込み操作が可能になる。

集中サーバアルゴリズムの欠点は、NRMB 戦略において、データ位置決め機構の説明で既に述べた通りである。

3. 固定分散サーバアルゴリズム (fixed distributed-server algorithm): この方式は、集中サーバ方式の直接の拡張である。この方式において、集中サーバの役割は、幾つかの分散サーバに分散されている。したがって、この方法では幾つかのノードにブロック管理者が存在する (図 5.14)。各ブロック管理者は、予め決められているブロックのサブセットを管理する。そして、ブロックと特定のブロック管理者及びそのノードを対応づけるマッピング機能を用いる。フォルトが発生すると、マッピング機能が現在の要求しているブロックを

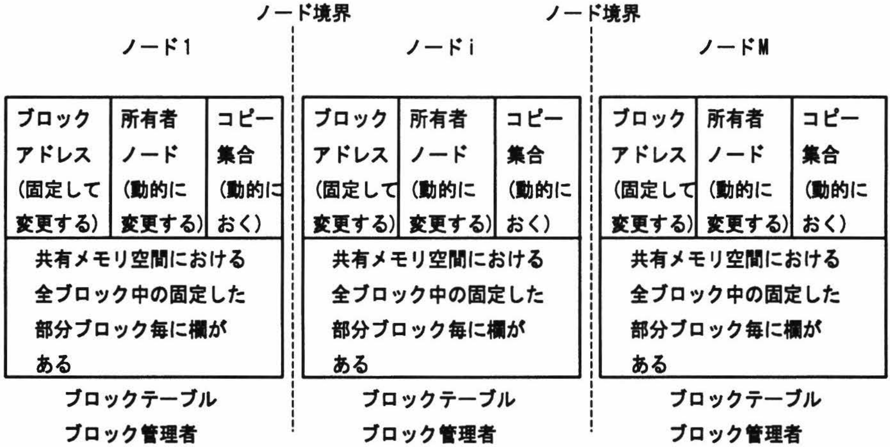


図 **5.14**　RMB 戦略のための固定分散サーバによるデータ位置位置決め機構におけるブロックテーブル

管理しているブロック管理者を見つけるのに使われる。そして、アクセスしているブロックへの要求は、そのノードのブロック管理者に送られる。要求は集中サーバアルゴリズムと同じ方法で扱われる。

固定分散サーバアルゴリズムの利点と制限は、NRMB 戦略でのデータ位置決め機構の記述で既に述べた通りである。

4. 動的分散サーバアルゴリズム (**dynamic distributed-server algorithm**):この方式は、NRMB 戦略の動的分散サーバアルゴリズムと似た方法で動作する。共有メモリ空間のすべてのブロックに対する欄を持つブロックテーブルを各ノードは持っている (図 5.15)。テーブルの各欄には、予想所有者フィールドを持ち、そこには対応するブロックの所有者の位置に関するヒントが記述されている。更に、もしノードがブロックの正しい所有者であるとき、そのノードのブロックテーブルのブロックに対する欄には、ブロックの正しいコピーを持つノードのリストを与えるコピー集合フィールドも含まれる。

フォルトが発生したとき、フォルトを発生したノード (N) のフォルトハンドラは、ローカルブロックテーブルのアクセスされたブロックに対する予想所有者ノード情報を取り出し、そのノードにブロック要求を送る。もし、そのノードがそのブロックの正しい所有者でないとき、そのノードはローカルのブロッ

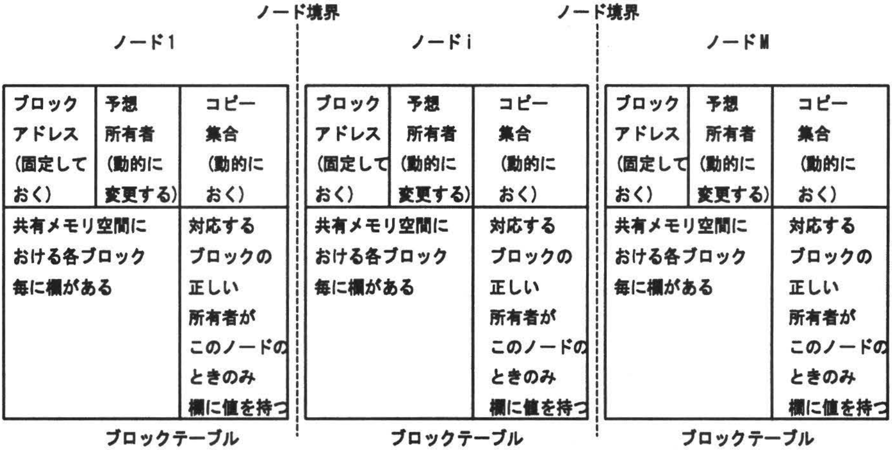


図 5.15 RMB 戦略のための動的分散サーバによるデータ位置特定機構におけるブロックテーブルの構造と位置

クテーブルを引いて、そのブロックの欄の予想所有者フィールドに示されているノードに要求を転送する。他方、ノードがそのブロックの正しい所有者であれば、そのノードは要求を次のように処理する。読み込みフォルトの場合、対応するブロックの欄のコピー集合フィールドに N を追加し、そしてブロックのコピーをノード N に送付する。そして、N は読み込み操作を実行する。書き込み操作の場合、そのブロックのコピーとコピー集合情報をノード N に送り、そしてローカルブロックテーブルからそのブロックのコピー集合情報を削除する。ブロックとコピー集合情報を受け取ると、ノード N はコピー集合に含まれるすべてのノードに無効化要求を送る。その後、ノード N はそのブロックの新しい所有者となり、書き込み操作の実行を進める。

ブロックの正しい所有者へ到達するまでのノードのチェーンの長さを削減するために、ブロックテーブルの予想所有者フィールドは以下のように更新される [Li and Hudak 1989]。

(a) ノードが無効化要求を受信したとき、
(b) ノードが所有権を手放したとき、つまり書き込みフォルトのとき
(c) ノードがフォルト要求を転送したとき、

最初の 2 つの場合は、予想所有者フィールドはそのブロックの新しい所有者に変更される。3 番目の場合は、予想所有者フィールドは元々のフォルトが発生したノード (N) に変更される。なぜなら、もし要求が書き込みならフォルトが発生したノード (N) は、新しい所有者になろうとしている。もし、要求が読み込みなら、要求が満足された後には、フォルトを発生したノードは、正しい所有者情報を持つことが分かっているからである。どちらの場合でも、そのブロックの予想所有者フィールドをノード N にすることは、良い考えである。

ブロックに対するどのノードでのフォルトも、そのブロックの正しい所有者に到達すること、そして全部で M ノードあったとき、最大 $M-1$ メッセージで正しいブロックの所有者を見つけることができることは [Li and Hudak 1989] に示されている。

上で述べたデータ特定アルゴリズムの幾つかの改良は、[Li and Hudak 1989] にある。

複製ありで、移動なしブロック

この戦略では、共有メモリブロックはシステムの複数のノードに複製されているが、その位置は固定されている。メモリアドレスに対する読み込み、書き込みアクセスは、そのメモリアドレスを含むブロックの複製を持っているノードの 1 つに送られる。ブロックのすべての複製は、書き込みアクセスの場合にすべてを更新することで一貫性を保つ。書き込み更新プロトコルは似たプロトコルがこの目的に使われる。順序一貫性は、すべてのノードへの書き込み操作を順序づけるために、グローバルシーケンサを使うことで保証している (図 5.11)。

RNMB 戦略におけるデータ位置決め: RNMB 戦略は、以下の特性を持っている。

1. ブロックの複製の位置は決して変化しない。
2. データブロックのすべての複製は、一貫性を持つ。
3. 読み込み要求についてのみ、その読み込み要求が実行されるメモリアドレスを含むブロックを持っている複製ノードの一つに直接送られる。書き込み要求は、最初にシーケンサに送られる。

これらの特徴に基づいて、この場合の読み込み／書き込み操作を扱うための

データ位置決めの最良の方法は、各ノードのブロックテーブルとシーケンサを使った順序テーブル (図 5.16) を持つことである。ノードのブロックテーブルは、共有メモリの各ブロックについての欄を持っている。各欄では、1 つのブロックがその複製の位置に対応づけられている。順序テーブルも、共有メモリの各ブロックに対応する欄を持っている。順序テーブルの各欄は 3 つのフィールドを持っている。それは、ブロックアドレスを持つフィールド、ブロックの複製を持っているノードのリストが入っている複製集合フィールド、そしてブロックに新しい更新が加わるたびに数字を 1 加える順序番号フィールドからなる。

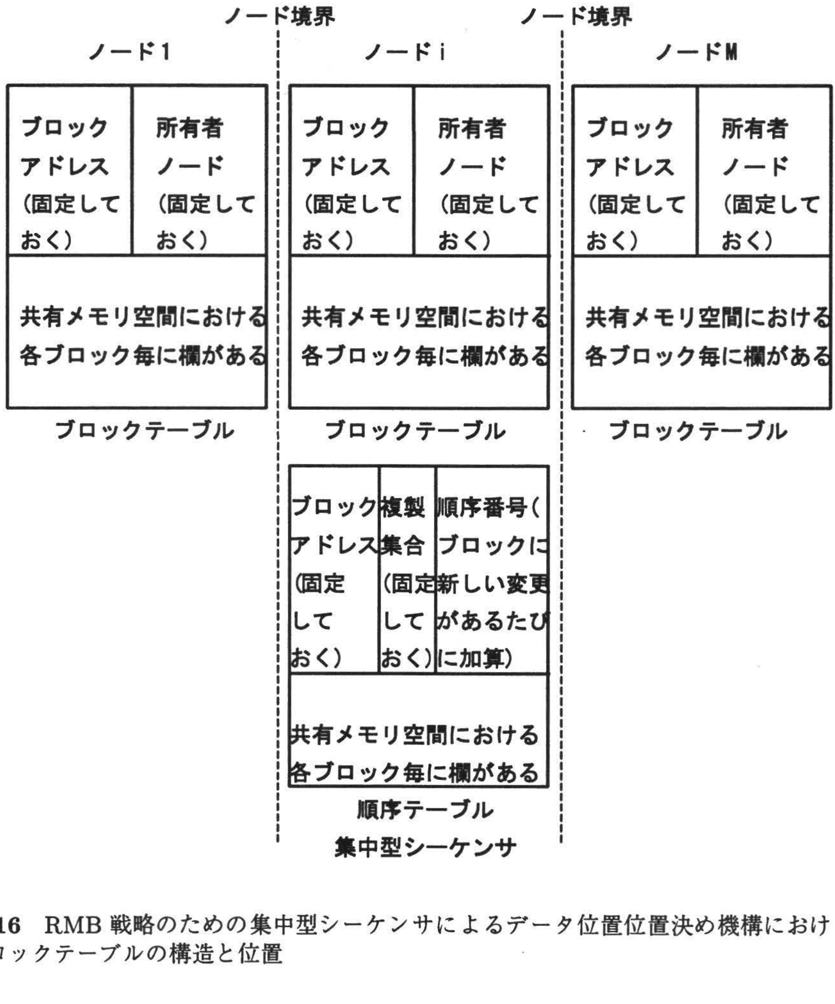


図 5.16 RMB 戦略のための集中型シーケンサによるデータ位置位置決め機構におけるブロックテーブルの構造と位置

ブロックに読み込み操作を行うために、ブロックの複製の位置がローカルブロックテーブルから取り出され、そして読み込み要求が直接そのノードに送られる。ブロックへの書き込み操作は、シーケンサに送られる。シーケンサは次の順序番号を変更要求に与える。シーケンサは、変更を順序番号を付けてこのブロックの複製集合フィールドにあるリストのノードにマルチキャストする。書き込み操作は、各ノードで順序番号順に実行される。

あるブロックに対するすべての読み込み操作が、同じ複製ノードに到着することを避けるために、異なるノードのブロックテーブルは異なる複製の位置が対応するブロックの欄に記入されなければならない。これは、異なるノードからの同じブロックへの読み込み操作が分散することに役立つ。

5.6.2 Munin：リリース一貫性をもつ DSM システム

一貫性モデルの議論において、順序一貫性に加えてリリース一貫性も DSM システムに利用できそうで魅力的な方法であることを見てきた。したがって、リリース一貫性 DSM システムの事例研究として Munin システムを以下に説明する [Bennett et al. 1991, Carter et al. 1994]。

共有メモリ空間の構造

Munin の共有メモリ空間は、**共有変数 (shared variables)**(プログラムデータ構造を含む) の集まりとして構造化される。共有変数は、キーワード shared によって定義され、コンパイラはそれを認識できる。プログラマは、標準的な注釈タイプ (共有変数に対する注釈のタイプについては後で述べる) を使って共有変数に注釈を付けることができる。各共有変数は、デフォルトでコンパイラによって別のページにおかれる。ページはネットワークを通して MMU ハードウェアがデータ転送をする単位である。しかし、プログラマは同じ注釈タイプを持つ複数の共有変数を同じページに配置することを指定することができる。同じページに異なる注釈タイプの変数を置くことは認められない。それは、ページに使われる一貫性プロトコルがそのページに含まれる変数の注釈タイプに依存するからである。明らかに、変数のサイズがページサイズより大きなものは、複数のページを占有する。共有変数はキーワード shared を使って宣言するため、コンパイラはそれを認識できる。

リリース一貫性の実装

リリース一貫性の説明において、リリース一貫性のためにはアプリケーションは臨界領域をモデル化する必要があることを見てきた。したがって、リリース一貫性をサポートするDSMシステムは、臨界領域を構成する機構とプログラミング言語が必要となる。Muninは2つのこのような同期機構を提供している。ロック機構とバリヤ機構である。

ロック機構は、lock同期変数とそれをアクセスするためのacquireLockとreleaseLockプリミティブを用いている。ロック変数をその引数に持つacquireLockプリミティブは、プロセスによって臨界領域に入るときに実行される。そして、同じロック変数を引数に持つreleaseLockプリミティブは、プロセスによって臨界領域を出るときに使われる。リリース一貫性を保持するため、共有変数への書き込み操作は臨界領域に入っているときのみ実行される。しかし、共有変数への読み込み操作は、臨界領域の外でも中でも実行できる。臨界領域でなされた共有変数への変更は、プロセスが臨界領域を出た時点で共有変数の複製を持っている他のノードへ送信される。もし、アプリケーションのプログラムが上で述べたように適当に構造化されているならば、このDSMシステムは順序一貫性を維持するように動作する。

プロセスが、ロック変数を獲得するためにacquireLock要求を実行すると、システムは最初にロック変数がローカルノードで利用可能かどうかを検査する。もし可能でないなら、予想される所有者機構が使ってロック変数の現在の所有者を見つける。そして、要求はそのノードに送られる。ロックがローカルにあるか遠隔にあるかによらず、もしそれが空いていれば、それを要求したプロセスに譲られる。空いていなければ、要求したプロセスはそのロック変数の獲得を待っているプロセスの待ち行列の最後に追加される。ロック変数が現在の所有者によって開放されると、それは待ち行列の次のプロセスに与えられる。

バリヤ機構は、barrier同期変数を使い、waitAtBarrierプリミティブをその変数をアクセスするのに使う。バリヤは集中バリヤサーバ機構を使って実現される。

ネットワークページフォルトでは、予想所有者に基づく動的な分散サーバアルゴリズム(既にRMB戦略の記述で説明した)がMuninで使われており、アクセスされた共有変数を捜すために使われる。コピーセット機構(これもRMB戦略の記述の中で説明した)に似た機構が複製ページの位置を追跡するのに使われる。

共有変数の注釈

Muninのリリース一貫性は、順序一貫性DSMシステムに比べてよりよい性能をアプリケーションに提供する。更なる性能の向上のためには、Muninは幾つかの共有変数に対する幾つかの標準的な注釈を定義し、各タイプに最も適切な異なる一貫性プロトコルを定義している。つまり、この方式における一貫性プロトコルは、個々のデータ項目の粒度で適用される。標準注釈と各タイプの変数に対する一貫性プロトコルは以下の通りである [Bennet et al. 1990]。

1. **読み込み専用 (read-only)**:読み込み専用と注釈されている共有変数は、不変データ項目である。これらの変数は、初期化された後に読み込みは行われるが書き込みは行われない。したがって、一貫性制御の問題は生じない。これらの変数は、決して変更されないので、それらがアクセスされるすべてのノードで自由に複製することができ、これらの平均読み込みコストは劇的に削減される。したがって、このような変数への参照がページフォルトを発生したときは、そのページを既に持っているノードのどれかからフォルトしたノードに変数を含むページをコピーする。読み込み専用変数は、MMUハードウェアによって保護され、このような変数に書き込みを行うとすると致命的な誤りとなる。

2. **移動性 (migratory)**:フェーズに分けてアクセスされる共有変数は、ここで各フェーズとは1つのプロセスによるアクセスの組に対応するが、移動性変数として注釈されるかもしれない。ロック機構は、移動性変数の一貫性を保つために使われる。これは、移動性変数がロック同期変数によって保護され、臨界領域で利用されるためである。

移動性変数にアクセスするためには、プロセスは最初に変数のロックを獲得する、次に変数を使い、そしてそれを使い終わるとロックを開放する。そのとき、システムは移動性変数に対してロックは1つのプロセスしか獲得できないようにしている。もし、プロセスが移動性変数のロックを獲得しようとしたときにネットワークページフォルトが発生すると、そのページが現在の所有者のノードからフォルトが発生したノードに移動されてくる。NRMB戦略がこの場合使われる。つまり、ページは必要に応じてあるノードから別のノードに移動される。しかし、ページは複製されない。したがって、システムには移動性変数の含まれるページは1つのコピーしか存在しない。

移動性変数は、それらに関係するロックを移動の際に統合することで効率的に扱うことである。つまり、ロックと変数は1つのメッセージとなってロックを獲得した次のプロセスの位置に移動する。

3. 書き込み共有 (write-shared):プログラマは、共有変数に対するこの注釈を、システムに対してこの変数が複数のプロセスによって並行して更新されるが、変数の同じ場所を更新することはないということを知らせるために使用することができる。例えば、行列において、異なるプロセスは異なる行／列の要素を更新する。ここで各プロセスは、1つの行／列の要素のみを更新する場合などである。Munin は書き込み共有変数の偽共有問題を避けるために、複数のプロセスによって並行に更新することを許している。

書き込み共有変数は、プロセスの共有があるすべてのノードに複製される。これは、この変数に対するアクセスがネットワークフォルトを発生した場合、その変数を含むページが現在存在しているノードからフォルトを発生したノードにコピーされる。もし、アクセスが書き込みアクセスの場合、システムは最初にページのコピーを作り (**双子ページ (twin page)** と呼ぶ)、そしてオリジナルのページを更新する。プロセスは、それをリリースする前に幾つかの書き込みを行ってよい。ページがリリースされると、システムはワード毎にオリジナルのページと双子ページを比較し、そして2つの差分をすべての複製を持つノードに送信する。

ノードが更新されたページの差分データを受け取ると、システムはローカルのページのコピーについても更新されているかどうか調べる。もし更新がなければ、ローカルのページのコピーは受信した差分を組み込むことで更新する。他方、もしページのローカルのコピーも更新があった場合、ローカルのコピー (双子の方) とその受信した差分データとをワード毎に比較する。もし、同じワードがローカルと遠隔の両方のコピーで更新されていない場合、オリジナルのローカルコピーを遠隔からの差分で更新する。他方、もし比較によってローカルと遠隔のコピーで同じワードの更新を検出した場合、矛盾が発生し、実行時誤りとなる。

4. 生産者消費者 (producer-consumer):1つのノードのプロセスのみが書き込んで (生産) そして、固定した他のプロセスの集合が読み込む (消費) 共有変数は、生産者消費者タイプと注釈される。Munin はこのタイプの変数には**イーガオブジェクト移動 (eager object movement)** 機構を使う。この機構では、消費者のノードが将来的に必要となるときに生産者のノードから消費者のノードに変数が送られる。そのため、消費者が変数にアクセスするときには、ネットワークページフォルトは発生しない。更に、書き込み更新プロトコ

ルが、生産者が変数を更新する場合には必ず変数の複製を更新するために使われている。もし必要なら、生産者はロック機構を使って複数の更新を一緒に送ることもある。この場合、生産者は同期ロックを獲得し、変数への幾つかの更新を行い、そしてロックを開放する。このとき、変数又は更新が消費者のプロセスのノードに送られる。

5. **結果 (result)**:結果変数は、複数のプロセスによって書き込まれ、1 つのプロセスのみが読み込むという観点で、生産者消費者変数とちょうど逆の変数である。異なるプロセスが変数の異なる部分に書き込むことは矛盾を生じない。この変数は、すべての部分が書き込まれたとき、読み込み専用となる。例えば、あるアプリケーションにおいて、行列のそれぞれの行／列の要素を生成して書き込む異なる**作業者 (worker)** プロセスがあり、一度行列が完成すれば“ マスタ ”プロセスが更にそれに処理を行うという場合がある。Munin は、更新がマスタプロセスを持つノードのみに到着しすべての変数の複製の位置 (各作業者プロセスのノードには変数の複製がある) には到着しない結果変数について、特殊な書き込み更新プロトコルを使っている。異なるプロセスからのこの変数への書き込みは矛盾しないので、すべての作業者プロセスは書き込み操作を並行して行うことが許される。更に、結果変数はすべての部分が書き込まれないと読み出されないために、作業者プロセスはそのプロセスが担当している部分の書き込みが完了した時点でのみ変数をリリースする。そのとき、すべての更新は 1 つのメッセージになりマスタプロセスのノードに送られる。

6. **還元 (reduction)**:原子的に共有変数を変更する必要がある場合、これを還元タイプと注釈する。例えば、並列計算のアプリケーションにおいて、全域的に最小の値は原子的に取り出され、もしローカルの最小の値よりも大きい場合には原子的に変更されなければならない。Munin では、還元変数は通常ロックされ (ロックの獲得)、読み込み、更新、そしてアンロック (ロックの開放) が行われる。より良い性能のため、還元変数は固定された所有者に格納される。この所有者は、他のプロセスからの変数への更新を受信し、異なるプロセスから受信した更新を同期させる。そして、変数を更新し、更新した変数をその複製の位置に送信する。

7. **既存 (conventional)**:上のタイプのどの注釈も付かない共有変数は既存変数である。既に述べた Munin のリリース一貫性プロトコルが、複製された既

存変数の一貫性を管理するのに使われる。この場合、書き込み無効化プロトコルが、既存変数の古い版をプロセスが読み込まないことを保証するのに使われる。既存変数を含むページは、変数に書き込みを実行したいプロセスの位置に動的に移動する。

Muninでの経験から、読み込み専用、移動的、そして書き込み共有の注釈タイプは、これらのタイプの変数が非常によく利用されるために便利である。しかし、生産者消費者、結果、そして還元の注釈タイプはこれらのタイプの変数が余り利用されないために余り有効ではない。

5.7 置き換え戦略

共有メモリブロックを動的に移動／複製することを許すDSMシステムでは、ノードでの共有データをキャッシュする可能なスペースが飽和したとき以下に示す問題を解決する必要がある。

1. 新しく必要になったブロックのスペースを作るためにどのブロックを置き換えるか。
2. どこに置き換えたブロックを配置するか。

5.7.1 どのブロックを置き換えるべきか

置き換えの問題は、ページ化された主記憶や共有メモリマルチプロセッサシステムに対して広く研究されてきた。通常の置き換えアルゴリズムの分類では、それらを以下のカテゴリに分けている [Smith 1982]。

1. 利用状況に基づくか基づかないか。利用状況に基づくアルゴリズムは、キャッシュ情報 (又はページ) の利用の履歴を追跡し、その情報を置き換えの決定に利用する。これはつまり、キャッシュ情報の再利用が自然にその情報の置き換えのステータスを上げていくというものである。**最新アクセス最小 (least recently used: LRU)** はこの種のアルゴリズムの例である。反対に、利用状況に基づかないアルゴリズムは、置き換えを行う際にキャッシュ情報の利用の記録を活用しない。先入れ先出し (FIFO) や Rand(ランダム、又は擬似ランダム) はこのクラスに属する。

2. 固定空間か可変空間か。固定空間アルゴリズムは、キャッシュサイズが固定であることを仮定しており、一方可変空間アルゴリズムは必要に応じてキャッシュサイズが動的に変更できることを仮定している。したがって、固定空間アルゴリズムは特定のキャッシュ情報の選択を簡潔に含んでいる。他方、可変空間アルゴリズムでは、取り出しが置き換えを意味せず、そしてスワップアウトは対応する取り出しなしに起こる。

可変空間アルゴリズムは DSM システムにとっては適当ではない。それは、仮想共有メモリのキャッシュとして動作する各ノードのメモリはサイズが固定されているからである。更に、利用状況に基づかないアルゴリズムに比較して、利用状況に基づくアルゴリズムがより DSM に適している。それは、データアクセスの局所性の利点を利用することを可能とするからである。しかし、ほとんどのキャッシュシステムと異なり、それらは簡単な LRU 方策を置き換えに用いているが、ほとんどの DSM システムはデータ項目の状態を変えて、優先度機構を用いている。一例として、IVY[Li 1986,1988] の DSM システムによって用いられている置き換え方策をここで述べる。IVY の DSM システムでは、ノードの各メモリブロックは以下の 5 つのタイプに分類される。

1. **未使用 (unused)**:現在使われていない空きメモリブロック。
2. **無効 (nil)**:無効化されたブロック。
3. **読み込み専用 (read-only)**:ノードが読み込みアクセス権のみを持つブロック。
4. **読み込み所有 (read-owned)**:ノードが読み込みアクセス権のみを持っているが、そのブロックの所有者であるブロック。
5. **書き込み可能 (writable)**:ノードが書き込みアクセス許可を持っているブロック。明らかに、このノードはブロックの所有者でもある。なぜなら、IVY は書き込み-無効化プロトコルを用いているからである。

このブロックの分類に基づいて、以下の置き換え優先度が利用される。

1. 未使用そして無効ブロックの両者が最高の置き換え優先度である。つまり、ブロックが必要な場合、これらが最初に置き換えられる。利用されていないブロックが最高の置き換え優先度を持つということは明らかである。無効ブロックも同じ優先度を持つ。それは、無効ブロックがもはや利用されることは

なく、そのブロックへの将来のアクセスはネットワークフォルトを発生するからである。無効ブロックは最近参照されたブロックであることもあり、これがまさに単純なLRU方策が適当でない理由である。

2. 読み込み専用ブロックは、次の置き換え優先度を持っている。これは、読み込み専用ブロックがその所有者に利用可能であり、そしてそのブロックを無視することが可能であるからである。そのノードが将来またそのブロックを要求するとき、そのブロックは所有者ノードからそのときに持ってくる必要がある。

3. その複製が他のノードに存在する読み込み所有及び書き込み可能ブロックは次の優先度を持つ。それは、所有権を複製のうちのどれかのノードに渡すのみで十分だからである。このブロックは、自身を送る必要はなく、小さなメッセージのみで良い。

4. このノードのみにコピーがある読み込み所有及び書き込み可能ブロックは、最小の優先度を持つ。それは、このようなブロックの置き換えは、ブロックの所有権と同時にブロックを現在のノードからどこかのノードに転送することを含んでいるからである。LRU方策は、ローカルキャッシュにあるすべてのブロックが同じ方策を持っているとき、置き換えるブロックを選択するのに使われる。

5.7.2 置き換えたブロックをどこに置くか

一度メモリブロックが置き換える対象として選択されると、もしそのブロックに有効な情報が何か残っている場合、それを失うべきでないことが保証されなければならない。例えば、未使用、無効又は読み込み専用の状態を持っているブロックを単純に捨てることは、データの消失にはつながらない。同様に、複製が他の幾つかのノードに存在している読み込み所有、又は書き込み可能ブロックについても害はない。しかし、他のノードに複製を持たない読み込み所有、又は書き込み可能ブロックを捨てることは、有効なデータを消失する可能性がある。したがって、捨てる前にそれらをどこかに格納するように注意することが必要である。置き換えが必要になった有効なブロックを格納するために、2つの一般に使われる方法を以下に示す。

1. 2次記憶の使用。この方法では、ブロックは単純にローカルディスクに転送される。この方法の利点は、メモリ空間を無駄にせず、もしこのノードが同じブロックを再度アクセスしようとしたとき、ネットワークアクセスなしにローカルでブロックを入手できる。

2. 他のノードのメモリ空間の使用。時には、ブロックをローカルディスクより転送するよりも、ネットワーク上を転送する方が速い場合がある。したがって、有効なブロックを格納するもう1つの方法として、システムのすべてのノードから空きメモリを追跡して、十分な空間があるノードのメモリへ置き換えブロックを単純に転送する方式がある。この方法は、他のすべてのノードの空きメモリ空間の表を管理する必要がある。この表は、各ノードが通常のトラフィックにメモリ状態情報をピギーバックして更新することもできる。

5.8　スラッシング

スラッシング (thrashing) は、システムがアプリケーションプロセスの実行についての有用な仕事に費やす時間に比較して、多くの時間を共有ブロックをあるノードから別のノードに転送することに費やす場合をいう。これは、データブロックをあるノードから別のノードへ移動することを認めている DSM システムとって、重大な性能問題を生じる。スラッシングは、以下の状況で発生する。

1. 2つ以上のノード上のプロセスによって実行されるインターリーブされたデータアクセスは、データブロックをあるノードから別のノードに移したり戻したりすることを素早く連続的に行う (**ピンポン効果**)。

2. 読み込み専用のパーミッションがついたブロックは、それが複製された直後に繰り返し無効化される。

このような状況は、参照の (ノード) 局所性が小さいことを示している。もし、不適切に扱えば、スラッシングはシステム性能をかなり低下させる。したがって、この問題を解決する手段を講じる必要がある。以下の方法が、DSM システムのスラッシング問題を解決するのに使われる。

1. アプリケーション制御によるロックの提供。短い時間だけ他のノードから

のデータのアクセスを拒絶するためのデータのロックは、スラッシングを減少させる。アプリケーション制御のロックは、この方法を実装するために各データブロックに対して行われる。

2. 最小時間ブロックをノードに固定。スラッシングを減少させるもう1つの方法は、あるブロックがそのノードに確保されてから、最小時間 t が経過するまではそのノードから取り去ることが許されないようにすることである。その時間 t は、静的に固定することもできるし、アクセスパターンに基づいて動的に変化させることもある。例えば、Mirage[Fleisch and Popek 1989] はこの方法をスラッシングを減少させるために使い、アクセスパターンに基づいてそのノードでブロックが利用できる最小時間を動的に決定した。

この方法の主な欠点は、適切な時間 t を決定することの難しさである。もし、この値が静的に固定されていると、多くの場合に不適切になってしまうことがある。例えば、もしプロセスがブロックを書き込みのために1度だけ使うとき、他のプロセスは時間 t が経過するまではアクセスすることが許されない。他方、もしプロセスがオブジェクトに対して幾つかの書き込み操作を行うとき、プロセスがそのブロックの使用を終了するまえに時間 t が経過してしまい、システムは他のプロセスにそのブロックをアクセスする権利を譲ってしまうかもしれない。したがって、値 t の動的な調整が好ましい方法である。この場合、あるブロックについての t の値は、そのブロックの過去のアクセスパターンに基づいて決定される。MMU の参照ビットがこの目的に使うことができる。そのブロックの値 t を決定するのに使われるもう1つの要因は、そのブロックにアクセスすることを待っているプロセスのキューの長さである。

3. 整合性アルゴリズムの共有データ利用パターンへの適応。スラッシングは、異なる特性を持った共有データに対する異なる整合性プロトコルを用いることでも最小化できる。例えば、Munin で書き込み共有変数に対して用いられている整合性プロトコルは、偽共有問題を回避することができる。そして、最終的にはスラッシングを回避する結果となる。

上の記述から、分散共有メモリの完全な透過性は、スラッシングを最小にする試みとの妥協であるといえる。それは、上で述べたほとんどの方法は、プログラマの補助を必要とするからである。例えば、アプリケーション制御のロックによる方法では、ロックの利用によって特定の共有メモリアルゴリズムに対して指示を与えることが必要となり、したがって共有メモリ抽象はもはや透過

ではない。更に、アプリケーションは、共有データにアクセスしていることやそのアクセスパターンについて注意を払う必要が生じる。同様に、Munin はプログラマに共有変数に対して標準的な注釈方法を使って注釈をつけることを要求している。これは、共有メモリ抽象を透過的でなくしている。

5.9 DSM についての他の方法

データキャッシュの管理 (置き方、そしてデータの移動) の方法に依存して、DSM システムを設計する 3 つの方法がある。

1. オペレーティングシステムによって管理されるデータキャッシュ
2. MMU ハードウェアによって管理されるデータキャッシュ
3. 言語実行時システムによって管理されるデータキャッシュ

この本がオペレーティングシステムの本であることから、この章では最初の方法に焦点をあてる。IVY[Li 1986] と Mirage[Fleisch and Popek 1989] はこの分類に属する。これらのシステムでは、各ノードは自分のメモリを持ち、他のノードのメモリにあるワードへのアクセスがオペレーティングシステムへのトラップを発生する。すると、このオペレーティングシステムは、その遠隔のマシンとメッセージを交換して、そのノードのメモリからアクセスされたワードを含むページを獲得する。したがって、これらのシステムでは、データの置き方と移動はオペレーティングシステムによって扱われる。議論の完全さのため、DSM システムの設計のための残りの 2 つの方法についても以下に簡単に述べる。

2 番目の方法は、MMU によってキャッシュを管理する方法である。この方法は、ハードウェアキャッシュを持つマルチプロセッサに用いられる。これらのシステムにおいて、DSM の実現はすべて、あるいはほとんどハードウェアでなされる。例えば、もしマルチプロセッサが単一のバスで接続されている場合、それらのキャッシュはバスを**スヌーピング (覗き見)** することで一貫性が保たれる。この場合、DSM はすべてハードウェアで実現される。DEC の Firefly ワークステーションはこの種類に属する。他方で、もしマルチプロセッサがスイッチで相互接続されている場合、どの CPU がどのキャッシュブロックを持っているかを追跡するために、ハードウェアキャッシュに加えてディレクトリが通常用いられる。キャッシュデータが一貫性を持つためのアルゴリズムは、主

にMMUのマイクロコードに格納される。したがって、この場合は、DSMはほとんどハードウェアで実現されている。スタンフォード大学のDash[Lenoski et al. 1992]とMITのAlewife[Agarwal et al. 1991, Kranz et al. 1993]システムがこの種類に属する。これらのシステムは、遠隔アクセスが検出されると、キャッシュコントローラあるいはMMU(オペレーティングシステムソフトウェアではなく)によってメッセージが遠隔のメモリに送られる。

3番目の方法は、言語実行時システムによってキャッシュを管理する方法である。これらのシステムでは、DSMは、0からすべてのマシンの結合された全メモリサイズまでの直接の線形なメモリとしてでなく、プログラミング言語の構造、つまり共有変数と(従来のプログラミング言語の)データ構造、あるいは共有オブジェクト(オブジェクト指向プログラミング言語)の集まりとして構造化される。これらのシステムでは、共有変数／オブジェクトの配置と移動は、オペレーティングシステムの協力のもと言語の実行時システムによって扱われる。これは、プロセスによって変数／オブジェクトがアクセスされたとき、要求された変数／オブジェクトへのアクセス操作を、現在の位置に関係なく成功裏に実行することが、言語実行時システムとオペレーティングシステムの責任である。この方法の利点は、プログラミング言語は、共有変数／オブジェクトのアプリケーションにおける利用パターンをプログラマが指定する機能を提供することもできるし、システムが幾つかの一貫性プロトコルをサポートして、共有変数／オブジェクトに最も適したものを選ぶこともできる。したがって、これらのシステムは別々のデータ項目の粒度で一貫性プロトコルを適用できるし、よりよい並行性のために順序一貫性モデルよりも弱い一貫性モデルに基づくこともできる。しかし、この方法の欠点は、プログラマに追加の負担を強いることである。Munin[Bennet et al. 1990]とMidway[Bershad et al. 1993]は、DSMの構造を共有変数とデータ構造の集まりとするシステムの例である。他方で、Orca[Bal et al. 1992]やLinda[Carriero and Gelernter 1989]はDSMを共有オブジェクトで構造化したシステムの例である。

5.10 不均一 DSM

異なるアーキテクチャのコンピュータは、通常異なる特徴的な機能を持っている。例えば、スーパーコンピュータとマルチプロセッサは計算主体のアプリケーションには良いが、一方でパーソナルコンピュータとワークステーションは通常良い利用者インタフェースを持っている。不均一の計算環境は、アプリ

ケーションにとって幾つかの異なるタイプのコンピュータの特徴的な機能の中で最良のものを選ぶことが可能となる。したがって、不均一性は、しばしば分散システムには望ましい性質である。

不均一 DSM は、不均一環境に共有メモリパラダイムを提供する機能であり、そして異なるアーキテクチャのマシン間でメモリ共有を可能とする。一見したところ、異なるアーキテクチャのマシン間の共有メモリはほとんど不可能に見える。しかし、Mermaid[Zhou et al. 1990,1992] と呼ばれた不均一分散システムの実験的なプロトタイプの測定によれば、不均一分散システムは利用可能であるばかりか、(少なくともあるアプリケーションでは) 同機種の対抗システムに匹敵することもある。不均一のマシンによるネットワーク上に DSM システムを構築するための 2 つの主な課題は、データ変換と、ブロックサイズの選択である。これらの問題を以下に述べていく。以下の記述は、[Zhou et al. 1990,1992] によって紹介された内容に基づいている。

5.10.1　データ変換

異なるアーキテクチャのマシンは、異なるバイトオーダや浮動小数点表記を用いているかもしれない。あるタイプのノードから別のタイプのノードにデータが送られるとき、宛先ノードでそれがアクセスされる前に変換されなければならない。しかし、DSM システムにおけるデータ転送はブロックである。したがって、異なるタイプのノード間でブロックの移動があるときは、ブロックの内容が変換される必要がある。しかし、変換はブロックに格納されたデータの型に基づいて行う必要がある。DSM システムが、ブロックに含まれるアプリケーションレベルのデータの型やブロックのレイアウトを知らずにブロックを変換することはできない。したがって、これはアプリケーションプログラマの助けが必要である。プログラマは、自分のアプリケーションプログラムによって使うメモリのレイアウトを知っている。以下に示すように、不均一 DSM のデータ変換に関する 2 つの方法が提案されている。

ソース言語オブジェクトの集まりとしての DSM システムの構造化

この方法では、DSM システムはソース言語の変数又はオブジェクトの集合として構造化されており、そのためデータ移動の単位はブロックではなくオブジェクトとなる。DSM コンパイラが、様々なマシンアーキテクチャ間でソース言語にある変数／オブジェクトを変換するルーチンを追加する。遠隔のノー

ドから共有メモリオブジェクトへの各アクセスに対して、DSM システムは要求しているノードとそのオブジェクトを保持しているノードが互換性があるかどうかを検査する。もし、互換性がなければ、オブジェクトを要求しているノードに移動される前に変換ルーチンが使われる。

このデータ変換の方法は、Agora 共有メモリシステム [Bisiani et al. 1987] で用いられている。Agora では、各マシンはマシンタグでメッセージをマークする。メッセージには、あるノードから他のノードへ移動されるオブジェクトが含まれている。同一の又は互換性のあるマシン間のメッセージは変換は必要ない。変換が必要な場合は、宛先の環境に関する情報の種類がオブジェクトの一方向変換の実行に使われる。フィールドタグとデータの整列のためのマシン要求に依存して、変換プロセスはバイトのフィールド毎の交換や、ギャップの挿入や削除、そしてビットの入れ替えなどの操作が含まれる。

ソース言語のオブジェクトの集合として DSM を構成することは、データ変換の観点からは便利であるが、性能の観点からは一般には推奨されない。それは、従来のプログラミング言語のオブジェクトの典型的なものがスカラー(単一のワード)、配列、そして構造体だからである。DSM システムの共有実体として、これらのタイプで適切なものはない。ここの共有実体に対して、各実体毎にアクセス権を設定する必要があり、そして 1 つの実体の移動についても通信オーバヘッドがかかる。したがって、共有の単位としてスカラーデータ型を選択することは、システムを非常に非効率なものにする。他方で、配列はデータの共有と移動にとっては扱いにくいほど大き過ぎることになりやすい。大きなデータオブジェクトは、偽共有とスラッシングを生じ易く、そしてその移動もトランスポートプロトコルのパケットサイズの制限のために、しばしば分割と組み立ての操作が必要となる。

唯一のデータタイプからなるブロック

この機構は、Mermaid[Zhou et al. 1990,1992] で用いられ、そのブロックサイズとしてページサイズを用いている。したがって、ページは 1 つのデータタイプしか含まない。Mermaid では、DSM ページ (ブロック) テーブルの欄には、そのページで保守されているデータのタイプと、そのページに確保されたデータの量を表す付加情報が保持されている。異なるアーキテクチャのマシン間でデータが移動されたときは、必ずそのページのデータを適切な形式に変換する。Mermaid はアプリケーションプログラマに対して、アプリケーションにおける各利用者定義データを変換するルーチン提供するよう要求している。

Mermaidの設計者は、ページの変換コストは、ページ移動に掛かるコストの全体に比べて小さいと指摘している。

Zhou et al.[1990] では、この方式の制限を以下のように述べている。

1. 1つのタイプだけを1つのページに含むことしか認めないのは、フラグメンテーションによってメモリを無駄にしてしまい、結果的にページングが増加することがある。

2. この機構は、異なるタイプのマシンで使われているコンパイラが、コンパイラ生成コード内において、各データタイプのサイズや複合構造におけるフィールドの順序が各マシンのもので同じでなければならないという意味で、互換性がなければならない。もし、アプリケーションレベルのデータ構造が2つのマシンで異なるといったような、互換性がないコンパイラを2つの異なるマシン用のコードを生成するのに利用する場合、2つのマシンにおけるページ間の対応は1対1にならない。これは、変換の後では1つの構造が1つのページに収まることができないこともあり、逆に、後続ページの幾らかのデータが現在のページを変換するときに必要となることもある [Zhou et al. 1990]。これは、変換プロセスを複雑化する。

3. この方法のもう1つの問題は、それが転送される前にアクセスされたのはほんの少しの量であるにも拘わらず、ページ全体を変換してしまうことである。ときには、この問題は最初の方法でも、オブジェクト全体を変換するために、重大な問題となりうる。例えば、大きな配列はメモリの幾つかのページに渡って占有し、そしてこのオブジェクトの移動は、移動の前にアクセスされたのが配列の先頭のほんの数個の要素だけだったにも拘わらず占有するページ全体を変換する。

4. この機構は完全には透過ではない。なぜなら、利用者に利用者定義のデータ型を変換するルーチンと、データタイプと変換ルーチンの対応を指定したテーブルの作成を要求しているからである。透過性の問題は、自動的な変換ルーチンの生成によって解決されるかもしれない。そして、対応テーブルは利用者プログラムのプリプロセッサによって解決されるかもしれない [Zhou et al. 1990]。

不均一 DSM システムでのデータ変換に関するもう1つの重大な問題は、数

値計算のアプリケーションでの浮動小数点数値の精度である。アプリケーションには、どれくらいデータを移動して変換するかについての制御を持たないため、データが数回変換されると浮動小数点データの数値計算上の精度が失われるかもしれず、その結果は数値計算上おかしなものになる可能性がある。

5.10.2 ブロックサイズ選択

同機種 DSM システムでは、ブロックサイズは通常元々の仮想メモリ (VM) のページと同じであり、そのため MMU ハードウェアが DSM のブロックフォルトを検出するのに利用できる。しかし、不均一環境においては、仮想メモリページは異なるタイプのマシンでは異なる可能性がある。そのため、ブロックサイズの選択は、このような状況では複雑な作業となる。Zhou et al.[1990,1992] は、不均一 DSM システムにおけるブロックサイズ選択のためのアルゴリズムとして以下の方式の利用を提案した。

1. 最大ページサイズアルゴリズム。この方法では、DSM ブロックサイズとしてすべてのマシンのうち最大の VM ページサイズを採用する。VM ページサイズは通常 2 のべき乗が使われるため、複数のより小さな VM ページは 1 つの DSM ブロックに一致する。もし、より小さなページサイズのノードでページフォルトが発生した場合、要求したページを含む複数のページのブロックが受信される。このアルゴリズムは、大きなサイズのブロックについて問題となる偽共有とスラッシングの問題が生じる。

2. 最小ページサイズアルゴリズム。この方法では、DSM ブロックサイズとしてすべてのマシンのうち最小の VM ページサイズを採用する。より大きなページサイズのノードでページフォルトが発生した場合、複数のブロック (このトータルサイズはフォルトしたノードのページサイズと同じサイズ) がページフォルトを満足するように移動する。このアルゴリズムはデータの競合を減少させるが、小さなサイズのブロックによる通信の増加と、ブロックテーブル管理のオーバヘッドの問題が生じる。

3. 中間ページサイズアルゴリズム。大きなサイズと小さなサイズのブロックが持つ問題の間でバランスをとるために、不均一 DSM システムではすべてのマシンの中で最大の VM ページサイズと最小のページサイズの間のどこかのブロックサイズを選択することもある。

5.11 DSMの利点

分散共有メモリは、疎結合システムの高レベルのプロセス間通信の機構である。メッセージパッシング機構を越えてこの方式が持つ利点のために、ますます注目を集めている。この利点について、以下で議論する。

5.11.1 より簡潔な抽象化

現在では、メッセージパッシングを使った疎結合分散メモリマシンの直接のプログラミングは、冗長であり誤りを生じ易いことが広く認識されている。この主な理由は、メッセージパッシングモデルではプロセスが通信プリミティブとチャネルやポートを明示的に利用しなければならないため、常にプログラマがプロセス間のデータの移動に注意していなければならないことによる。この労力を軽減するために、手続き呼び出しのインタフェースを提供するRPCが導入された。しかし、RPCでさえ、呼び出し側のアドレス空間と異なるアドレス空間で手続き呼び出しが実行されるために、呼び出し側が文脈依存のデータや複雑なデータ構造を渡すことは困難である。つまり、パラメータは値で渡さなければならない。メッセージパッシングモデルでは、プログラミング作業はプロセス間でメッセージの形式で渡される複雑なデータ構造をパックしたりパックを解いたりすることによって、更に複雑になる。共有メモリプログラミングのパラダイムでは、プログラマをこのような低レベルの作業の多くから隔離する。したがって、DSMの主な利点は、疎結合分散共有メモリマシンのアプリケーションプログラマに提供するための簡潔な抽象化である。

5.11.2 分散アプリケーションプログラムの可搬性の向上

DSMに用いられているアクセスプロトコルは、シーケンシャルなアプリケーションがデータをアクセスする方法と一貫性がある。これは、シーケンシャルなものから分散アプリケーションへのより自然な移行を可能とする。基本的に、共有メモリマルチプロセッサシステムに対して記述された分散アプリケーションプログラムは、変更無しに分散共有メモリシステム上で動作することができる。したがって、既存の分散アプリケーションプログラムをDSM機能を持った分散メモリシステムに移植することはDSM機能を持たない分散メモリシステムに移植するよりも容易である。

5.11.3 同じアプリケーションの性能の向上

DSM抽象を提供するソフトウェアのレイヤは、メッセージパッシングシステムの上に実現され、下位のメッセージパッシング通信システムのサービスを使う。したがって、基本的にDSMを使っているアプリケーションの性能は、それらが直接メッセージパッシングサービスを使っている場合に比べて悪くなることが予想される。しかし、これは常に正しくはなく、DSMを使っている幾つかのアプリケーションではメッセージパッシングを使っているものよりも良い性能が出る場合もあることが分かっている。これは、次の3つの理由から可能となる[Stumm and Zhou 1990]。

1. データの局所性。DSMの計算モデルは、データを動き回らせることによって、よりアクセスしやすくする。DSMアルゴリズムでは、通常大きなブロックでノード間でデータを移動する。従って、データアクセスに合理的な程度のデータの局所性があるアプリケーションでは、通信オーバヘッドは複数のメモリアクセスと比べて減償できるものとなる。

2. 必要に応じたデータ移動。DSMの計算モデルは、データがアクセスされる必要が生じた時点でのデータの移動をも可能とする。他方では、幾つかの分散アプリケーションではフェーズに分けて実行されるものがある。フェーズとは、データ交換フェーズが前にあって、計算フェーズが続くようなものである。データ交換フェーズで必要とされる時間は、しばしば既存の通信ボトルネックのスループットによって計算される。したがって、このようなアプリケーションでは、DSMによって提供される必要に応じたデータ移動の機能がデータ交換フェーズを削減し、通信の負荷を長い期間に分散させ、そして並行性の度合いを上げることに役立つ。

3. より大きいメモリ空間。DSM機能を使えば、全体のメモリサイズはシステムにあるノードのメモリサイズの合計となる。したがって、ディスクアクセスを含むページングやスワッピングの作業が非常に軽減される。

5.11.4　柔軟な通信環境

メッセージパッシングパラダイムでは、受信者の特定や送信者プロセスと受信者プロセスの共存が必要である。これはつまり、1 つのデータを送る送信プロセスは、その受信プロセスの名前を知らなければならない (マルチキャスト通信は除く)、そしてそのデータの受信者はデータが送信されるときに存在していなければならず、そして (最後には) 受信できる状態になければならない。さもなければ、データは配送されない。対称的に、DSM の共有メモリパラダイムは、送信者はそのデータの受信プロセスの名前を指定することがなく、より柔軟な通信環境を提供する。送信者は単純に共有メモリにデータを置き、そして受信者はそれを共有メモリから直接アクセスする。そのため、送信者と受信者の共存についても、共有メモリパラダイムでは必要ではない。実際、共有データのライフタイムは、どの受信プロセスのライフタイムとも独立である。

5.11.5　プロセス移動の容易性

分散システムにおいて、あるノードから他のノードにプロセスを移動させること (8 章にて述べた) は、移動されるプロセスのアドレス空間を前のノードから新しいノードに転送する必要があるため、手間がかかり時間が掛かる作業であることが分かっている。しかし、DSM の計算モデルでは、プロセッサ間の必要に応じたデータ移動の機能が提供されている。この機能によって、移動されるプロセスは移動の時にそのアドレス空間を前のノードに置いたままにして、新しいノードにおいてアクセスするのに必要になった時点で必要なページを取ってくればよい。したがって、DSM 機能を持つ分散システムでは、プロセス移動は前のノードのプロセッサから移動プロセスの**プロセス制御ブロック (PCB)** を取り除き、新しいノードのプロセッサのレディーキューに追加するだけの簡単な処理となる。PCB は、プロセス状態や CPU レジスタ、スケジューリング情報、メモリ管理情報、I/O 状態情報等の有用な情報を含む、プロセス毎に存在するデータブロック、又はレコードである。この方法は、分散システムにおけるプロセッサ間のプロセス移動についての非常に自然で効率的な形式を提供している。

5.12 まとめ

粗結合分散メモリマシンに対して、メッセージパッシングパラダイムによってプログラミングすることは難しく誤りが発生しやすい作業である。分散共有メモリ (DSM) 機能は、プログラマに対して共有メモリパラダイムを使ったプログラムを書くことを可能とし、高位の抽象化を提供することでこのプログラミング作業を容易にしている。また、この抽象化は逐次的なアプリケーションがデータをアクセスするのと整合性を持っている。

DSM システムの設計と実装に含まれる重要な問題は、データ共有の粒度、共有メモリ空間の構造、メモリ一貫性とアクセス同期、データの配置とアクセス、置き換え戦略、スラッシングの扱い、そして不均一性である。

粒度とは、ブロックサイズを指し、データ共有のそしてネットワーク上のデータ転送の単位である。大きなサイズ、そして小さなサイズのブロックのどちらにも利点と限界がある。幾つかの DSM システムは、ブロックサイズとして仮想メモリのページサイズを選択している。そのため、MMU ハードウェアが DSM ブロックフォルトのトリガとして利用されている。

DSM システムの共有メモリ空間の構造は、そのシステムがアプリケーションプログラマに見せる抽象的なビューを定義している。DSM システムの共有メモリ空間を構造化するためによく使われる 3 つの方法は、構造化なし、データタイプによる構造化、そしてデータベースとしての構造化である。DSM システムの構造と粒度は深く関連している。

メモリ一貫性は、DSM システムの設計にとって重要な問題である。これは、システム内の 2 つ以上のノードの主メモリ上に存在するデータブロックの一貫性を扱っている。この問題を扱うために幾つかの一貫性モデルがこの世界で提案されてきた。この章で述べた主なものは、厳密一貫性、順序一貫性、因果一貫性、PRAM 一貫性、プロセッサ一貫性、弱一貫性、そしてリリース一貫性である。この中で順序一貫性とリリース一貫性は多くのアプリケーションに適している。

DSM システムに順序一貫性を実現するプロトコルは、DSM システムに使われる以下の 4 つの複製と移動戦略に依存している。

1. ブロックの複製なしで、移動なし (NRNMB)
2. ブロックの複製なしで、移動あり (NRMB)
3. ブロックの複製ありで、移動あり (RMB)

4. ブロックの複製ありで、移動なし (RNMB)

これらの各戦略に対して、順序一貫性を実現するプロトコルはこの章で述べた。各場合に対して、適切なデータ配置機構についても述べた。加えて、リリース一貫性のDSMシステムの例として、Muninシステムについても説明した。

置き換え戦略は、共有データをキャッシングするためのスペースが一杯になったときに、置き換えるべきブロックの選択と、置き換えたブロックの置き場所についてを扱う。置き換えアルゴリズムの一般的な分類は、利用度に基づく方法か、基づかない方法か、固定空間か可変空間かがある。その置き換えの際に、利用度が高いブロックはローカルノードの2次記憶又は他のノードのメモリ空間にも置かれる。

スラッシングとは、1つのノードが他のノードにデータブロックを移動できるDSMシステムにとって、重大な性能上の問題となる。DSMシステムにおけるスラッシング問題を解決するために使われる方法は、アプリケーション制御ロック、最小時間ブロックをノードに束縛する方法、共有データの利用パターンに合わせて一貫性アルゴリズムを設計する方法が提供されている。

データキャッシングを管理する方法によって、DSMシステムを設計する3つの主な方法がある。すなわち、オペレーティングシステムによるキャッシュ管理、MMUハードウェアによるデータキャッシュ、言語実行時システムによるキャッシュの管理である。

不均一DSMは、不均一環境における共有メモリパラダイムを提供し、異なるアーキテクチャのマシン間でメモリ共有を行うための機構である。この機構を構築する上での2つの主な問題は、データ変換とブロックサイズの選択である。不均一DSMにおけるデータ変換に対して、従来提案されてきた2つの方法は、DSMシステムをソース言語のオブジェクトの集まりとして構造化する方法と、DSMブロック内に1つのタイプしか認めない方法である。不均一DSMシステムにおけるブロックサイズによく用いられている3つのアルゴリズムは、最大ページサイズアルゴリズム、最小ページサイズアルゴリズム、そして中間ページサイズアルゴリズムである。

DSMシステムは、実働できることをそしてそれがメッセージパッシングシステムより幾つかの利点をもっていることを研究が示してきた。しかし、それらがまだ成熟しているとはいえない。ほとんどの現存するDSMシステムは数個のノードからなる非常に小さな実験又はプロトタイプシステムである。性能結果は予備的な段階のものであり、少数のアプリケーション又は合成した負荷

に基づくものである。しかしながら、研究によってDSMは並列処理を効果的にサポートし、これからの10年余において成果の期待される活気ある研究分野である。

5.13 演習問題

5.1 分散共有メモリ抽象は、下位のメッセージパッシング通信システムのサービスを使って実現されている。そのため、基本的にDSMを使ったアプリケーションの性能はそれが直接メッセージパッシングを使った場合より悪くなると考えられる。この事実にも拘わらず、なぜ分散オペレーティングシステムの設計者の幾人かは彼らのシステムにDSMをサポートしているのか。DSMを使ったシステムの性能がメッセージパッシングのみを使ったシステムより良い性能を持つアプリケーションがあるか。もしあるならそのようなアプリケーションの種類について述べよ。もしないなら、なぜないかを説明せよ。

5.2 ブロック単位のDSMシステムの設計において、大きなブロックサイズを用いた場合と、小さなブロックサイズを用いた場合での相対的な利点、欠点を議論せよ。なぜ、ほとんどのDSMシステムの設計者はDSMのブロックサイズとして既存の仮想メモリの実装で使われる典型的なページサイズを好んで用いるのか。

5.3 共有メモリ空間の構造と、DSMシステムにおけるデータ共有の粒度とは密接に関連しているといわれる。理由を説明せよ。

5.4 偽共有とは何か。いつそれはおこるのか。この問題はDSMシステムの他の問題を引き起こすことになるか。読者の回答とその理由を述べよ。

5.5 偽共有問題を最小にするには何をする必要があるか。この問題を完全に解決できるか。もし誰かが偽共有問題を解決したとき、どんな他の問題が起こりうるか。

5.6 DSMシステムの設計において、NRNMB, NRMB、RMB、そしてRNMB戦略を用いるときの相対的な利点と欠点を議論せよ。

5.7 NRMB戦略を使っているDSMシステムで使われる各種のデータ配置機構の相対的な利点、欠点を議論せよ。

5.8 順序一貫性によるDSMシステムは、RMB戦略を用いている。その方式では、データブロックの更新にwrite-invalidate方式を用いており、ブロックの所有者を見つけるとき、そしてブロックの正しいコピーを現在持っているノードを追跡するのに集中サーバアルゴリズムを使っている。このDSMシス

テムのメモリ一貫性を実現する疑似コードを書け。

5.9 write-update プロトコルを用いている順序一貫性の DSM システムでは、なぜグローバルシーケンサが必要なのか。

5.10 オペレーティングシステムによってキャッシュが管理されているほとんどの DSM システムでは、一貫性に対して write-update の代わりに write-invalidate を使っている。なぜかを説明せよ。

5.11 弱一貫性とリリース一貫性の違いを述べよ。DSM システムの設計に対して、2 つのうちどちらを使うか。その回答と理由を述べよ。

5.12 あるプログラマがリリース一貫性の DSM システムを使ったアプリケーションを書いている。しかし、そのアプリケーションは正しい結果を出すために順序一貫性が必要である。このプログラマはどのような用心をする必要があるか。

5.13 PRAM 一貫性とプロセッサ一貫性の違いを述べよ。

5.14 順序一貫性とリリース一貫性モデルとの相対的な利点、欠点を述べよ。

5.15 因果一貫性とは何か。因果一貫性が最も適している一貫性モデルである応用例を示せ。

5.16 共有メモリ空間がオブジェクトとして構造化されている DSM システムに適している置き換えアルゴリズムを提案せよ。この場合の目標の一つは、メモリフラグメンテーションを最小化することである。

5.17 DSM システムのブロックを置き換える置き換え方策として、バッファキャッシュの中のキャッシュラインを置き換えるために使われる LRU 方策が良く働かないのはなぜか。

5.18 置き換えられたブロックをどこに置くかという問題を扱うために Memnet[Delp 1988] は、“ホームメモリ”という概念を導入し、各ブロックにホームメモリを持たせるようにした。ブロックの置き換えによってそのブロックがどこかのノードのメモリに転送されることが必要になったとき、そのブロックのホームメモリがあるノードに転送される。この章で述べた方式と比較して、この方法はどのような利点、欠点があるか。

5.19 DSM システムのスラッシングの主な原因は何か。DSM システムにおけるスラッシング問題を解決するために一般に用いられている方法は何か。

5.20 スラッシングを最小にするために、ある方法が用いられるとき、DSM システムの完全な透過性が妥協させられる。そのため、DSM システムの設計者は、スラッシングを最小にする方法を用いるより、システムがこの問題を自動的に検出しそして解決する方法を用いるべきとする者もいる。システムがス

ラッシングを検出し、一度検出するとそれを解決する方法を提案せよ。

5.21 ある分散システムは、DSM機能を持っている。このシステムのプロセススケジューリング機構は、現在動作中のプロセスがページフォルトを発生したときに、もう1つのプロセスを選択して動作させる。そして、ブロックがフェッチされている間もCPUは使用されている。このシステムのCPUをどのように利用するかについて議論している二人のシステムエンジニアは以下の意見を持っている。

(a) 最初の一人の意見は、もしたくさんのプロセスが1つのノードで実行されるようにスケジュールされると、そのノードで利用できるメモリ空間はそれらのプロセスに分散されるため、ページフォルトが起こったとき、ほとんど常に実行可能なプロセスが存在していることになる。したがって、CPUの利用率は高く保たれる。

(b) 二人目の意見は、少しのプロセスが1つのノードでの実行のためにスケジュールされると、そのノードの利用できるメモリ空間は少しのプロセスの各々に割り当てられ、各プロセスはより少ないページフォルトしか発生しなくなる。したがって、CPU利用率は高く保たれる。

誰の主張が正しいか、回答とその理由を述べよ。

5.22 DSMシステムを設計する3つの主な方法は何か。

5.23 不均一の計算機ネットワーク上にDSMシステムを構築するときの幾つかの問題は何か。これらの問題を扱う適切な方法を示せ。

5.24 LANとWANの環境の両者にとって、DSMシステムは適切か。回答とその理由を示せ。

5.25 DSMシステムにおいて、ネットワークブロックフォルトを削減するためのプログラミング手法について述べよ。

5.26 以下のタイプのDSMシステムについて、ブロックフォルトを扱うための疑似コードを書け。

(a) NRNMB戦略を用いたDSMシステム

(b) NRMB戦略を用いたDSMシステム

(c) RMB戦略を用いたDSMシステム

(d) RNMB戦略を用いたDSMシステム

必要なら幾つかの仮定をおいてよい。ただし、使った仮定は示すこと。

参考文献

[Adve, S., and Hill 1990] Adve,S., and Hill,M., "Weak Ordering: A New Definition," In:*Proceedings of the 17th International Symposium on Computer Architecture*, Association for Computing Machinery, New York, NY, pp.2-14 (1990).

[Agarwal et al. 1991] Agarwal, A., Chaiken, D., D'Souza, G., Fohnson, K., Kranz D., Kubiatowicz, J., Kurihara, K., Lim, B., Maa, G., Nussbaum, D., Parkin, M., and Yeung, D., "The MIT Alewife Machine: A Large-Scale Distributed Memory Multiprocessor," In: *Proceedings of the Workshop on Scalable Shared Memory Multiprocessors*, Kluwer Academic, Norwell, MA (1991).

[Bal 1991] Bal, H. E., Programming Distributed Systems, Prentice-Hall, London, England (1991).

[Bal et al. 1992] Bal, H. E., Kaashoek, M. F., and Tanenbaum, A. S., "Orca: A Language for Parallel Programming of Distributed Systems," *IEEE Transactions on Software Engineering*, Vol.SE-18, pp. 190-205 (1992).

[Baldoni et al. 1995] Baldoni, R., Mostefaoui, A., and Raynal, M., "Efficient Causally Ordered Communications for Multimedia Real Time Applications," In: *Proceedings of the 4th International Symposium on High Performance Distributed Computing*, IEEE, New York (1995).

[Bennett et al. 1990] Bennett, J., Carter, J., and Zwaenepoel, W., "Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence," In: *Proceedings of the 1990 Conference on Principles and Practice of Parallel Programming*, Association for Computing Machinery, New York, pp. 168-176 (1990).

[Bershad et al. 1993] Bershad, B. N., Zekauskas, M. J., and Sawdon, W. A.m "The Midway Distributed Shared Memory System," In: *Proceedings of the IEEE COMPCON Conference*, IEEE, New York, pp. 528-537 (1993).

[Bisiani and Ravishankar 1990] Bisiani, R., and Ravishankar, M., "Plus: A distributed Shared-Memory System," In: *Proceedings of the 17th International Symposium on Computer Architecture*, Association for Computing Machinery, New York, NY, pp. 115-124 (1990).

[Bisiani et al. 1987] Bisiani, R., Alleva, F., Correrini, F., Forin, A., Lecouat, F., and Lerner, R., "Heterogeneous Parallel Processing: The

Agora Shared Memory," Technical Report No. CMU-CS-87-112, Computer Science Department, Carnegie-Mellon University (Marchi 1987).

[**Bisiani et al. 1989**] Bisiani, R., Nowatzyk, A., and Ravishankar, M., "Coherent Shared Memory on a Distributed Memory Machine," In: *Proceedings of the International Conference on Parallel Processing*, IEEE, New York, pp. 1333-141 (August 1989).

[**Campine et al. 1990**] Campine, G. A., Geer, Fr., D. E, and Ruh, W. N., "Project Athena as a Distributed Computer System," *IEEE Computer*, Vol. 23, pp, 40-51 (1990).

[**Carriero and Gelernter 1989**] Carriero, N., and Gelerter, D., "Linda in Context," *Communications of the ACM*, Vol. 32, No. 4, pp.444-458 (1989).

[**Carriero et al. 1986**] Carriero, N., Gelernter, D., and Leichter, J., "Distributed Data Structures in Linda," In: *Proceedings of the ACM Symposium on Principles of Programming Languages*, Associaation for Computing Machinery, New York (1986).

[**Carter et al. 1991**] Carter, J. B., Bennett, J. K., and Zwaenepoel, W., "Implementation and Performance of Munin," In: *Proceedings of the 13th Symposium on Operating Systems Principles*, Association for Computing Machinery, New York, NY, pp.152-164 (1991).

[**Carter et al. 1994**] Carter, J. B., Bennett, J. K., and Zwaenepoel, W., "Techniques for Reducing Consistency-Related Communication in Distributed Shared Memory Systems," *ACM Transactions on Computer Systems*, Vol. 12 (1994).

[**Chase et al. 1989**] Chase, J. S., Amador, F. G., Lazowska, E. D., Levy, H. M., and Littlefield, R. J., "The Amber System: Parallel Programming on a Network of Multiproessors," In: *Proceedings of the 12th Symposium on Operating Sysgtems Principles*, Association for Computing Machinery, New York, NY, pp. 147-158 (1989).

[**Cheong and Veidenbaum 1988**] Cheong, H., and Veidenbaum, A. V., "A Cache Coherence Scheme with Fast Selective Invalidation," In: *Proceedings of the 15th International Symposium on Computer Architecture*, Association for Computing Machinery, New York, NY, pp. 299-307 (1988).

[**Cheriton 1986**] Cheriton, D. R., "Problem-Oriented Shared Memory: A Decentralized Approachi to aDistributed System Design," In: *Proceedings*

of the 6th International Conference on Distributed Computing Systems, IEEE, New York, pp. 190-197 (May 1986).

[Cheriton et al. 1991] Cheriton, D. R., Goosen, H. A., and Boyle, P. D., "paradigm: A Highly Scalable Shared-Memory Multicomputer Architecture," *IEEE Computer*, Vol. 24, No. 2, pp. 33-46 (1991).

[Cox and Fowler 1989] Cox, A. L., and Fowler, R. J., "The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experiences with PLATINUM," In: *Proceedings of the 12th Symposium on Operating Systems Principles*, Association for Computing Machinery, New York, NY, pp. 32-34 (December 1989).

[Dasgupta et al. 1991] Dasgupta, P., Leblanc, R.j., Ahmad, Jr., M., and Ramachandran, U., "The Clouds Distributed Operating System," *IEEE Computer*, Vol. 24, No. 11, pp. 34-44 (1991).

[Delp 1988] Delp, G. S., "The Architecture and Implementation of Mem-Net: An Experiment on High-Speed Memorhy Mapped Network Interface," Ph.D. dissertation, Department of Computer Science, University of Delaware (1988).

[Delp et al. 1991] Delp, G. S., Farber, D. J., Minnich, R. G., Smith, J. M., and Tam, M. C., "Memory as a Network Abstraction," *IEEE Network*, Vol. 5, pp. 34-41 (1991).

[Dubois et al. 1986] Dubois, M., Scheurich, C., and Briggs, F. A., "Memory Access Buffering in Multiprocessors," In: *Proceedings of the 13th Annual Symposium on Computer Architecture*, Association for Computing Machinery, New York , NY, pp. 343-442 (1986).

[Dubois et al. 1988] Dubois, M., Scheurich, C., and Briggs, F. A., "Synchronization, Coherence, and Event Ordering in Multiprocessors," *IEEE Computer*, Vol. 21, No. 2, pp. 9-21 (1988).

[Fekete et al. 1995] Fekete, A, Kaashoek, F., and Lynch, N., "Providing Sequentially-Consistent Shared Objects Using Group and Point-to-Point Communication," In: *Proceedings of the 15th International Confrence on Distributed Computing Systems*, IEEE, New York (May-June 1995).

[Fleisch 1987] Fleisch, B. D., "Distributed Shared Memory in a Loosely Coupled Distributed System," In: *Proceedings of the 1987 ACM SIGCOMM Workshop*, Association for Computing Machinery, New York, NY (1987).

[Forin et al. 1989] Forin, A., Barrera, J., Young, M., and Rashid, R., "Design, Implementation, and Performance Evaluation of a Distributed Shared Memory Server for Mach," In: *Proceedings of the 1989 Winter Usenix Confernce* (January 1989).

[Frank 1984] Frank, S. J., "Tightly Coupled Multiprocessor System Speeds Memory-Access Times," *Electronics*, pp. 164-169 (January 1984).

[Garcia Molina and Wiederhold 1982] Garcia-Molina, H., and Wiederhold, G., "Read-Only Transactions in a Distributed Database," *ACM Transactions on Database Systems*, Vol. 7, No. 2, Association for Computing Machinery, New York ,NY, pp. 209-234 (1982).

[Gharachorloo et al. 1990] Gharachorloo. K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A.,and Hennessy, J., "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors," In: *Proceedings of the 17th Annual Symposium on Computer Architecture*, Association for Computing Machinery, New York, NY¡ pp. 15-26 (1990).

[Gharachorloo et al. 1991] Gharachorloo, K., Gupta, A., and Hennessy, J., "Performance Evaluation of Memory Consisutecdy Models for Shared-Memory Multiprocessors," In: *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, IEEE Computer Society Press, Los Alamitos, CA, pp.245-257 (1991).

[Ghose 1995] Ghose, K., "SNOW: Hardware Supported Distributed Shared Memory over a Network of Workstations," In: *Proceedings of the 24th Annual International Conference on Parallel Processing*, IEEE, New York (August 1995).

[Goodman 1983] Goodman, J. R., "Using Cache Memory to Reduce Processor-Memory Traffic," In: *Proceedings of the 10th Annual Symposium on Computer Architecture*, Association for Computing Machinery, New York, NY, pp. 124-131 (June 1983).

[Goodman 1989] Goodman, J. R., "Cache Consistency and Sequential Consistency," Technical Report No. 61, IEEE Scalable Coherent Interface Working Group, IEEE, New York (1989).

[Goodman et al. 1989] Goodman, J. R., Vernon, M. K., and Woest, P. J., "Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors," In: *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, IEEE Compugter Society Press, Los Alamitos, CA, pp. 64-73 (1989).

[Harty and Cheriton 1992] Harty, K., and Cheriton, D., "Application-Controlled Physical Memory Using External Page-Cache Management," In: *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, Association for Computing Machinery, New York NY, pp. 187-199 (1992).

[Hutto and Ahamad 1990] Hutto, P. W., and Ahamad, M., "Slow Memory: Weakening Consistency to Enhance Concurrency in Distributed Shared Memories," In: *Proceedings of the 10th International Conference on Distributed Computing Systems*, IEEE, New York, pp. 302-311 (1990).

[Johnson et al. 1995] Johnson, D., Lilja, D., and Riedl, J., "A Circulating Active Barrier Synchronization Mechanism," In: *Proceedings of the 24th Annual International Conference on Parallel Processing*, IEEE New York (August 1995).

[Katz et al. 1985] Katz, R. H., Eggers, S. J., Wood, D. A., Perkins, C. L., and Sheldon, R. G., "Implementing a Cache Consistency Protocol," In: *Proceedings of the 12th Annual Symposium on Computer Architecture*, Associatoin for Computing Machinery, New York, NY, pp. 276-283 (June 1985).

[Keleher et al. 1992] Keleher, P., Cox, A. L., and Zwaenepoel, W., "Lazy Release Consistency," In: *Proceedings of the 19th International Symposium on Computer Architecture*, Association for Computing Machinery, New York, NY, pp. 13-21 (1992).

[Kessler and Livny 1989] Kessler, R. E., and Livny, M., "An Analysis of Distributed Shared Memory Algorithms," In: *Proceedings of the 9th International Conference on Distributed Computing Systems*, IEEE, New York, pp. 98-104 (June 1989).

[Kranz et al. 1993] Kranzk, D., Johnson, K., Agarwal, A., Kubiatowicz, J. J., and Lim, B., "Integrating Message Passing and Shared Memory: Early Experiences," In: *Proceedings of the 4th Symposium on Principles and Practice of Parallel Programming*, Association for Computing Machinery, New York, NY, pp. 54-63 (May 1993).

[Lamport 1979] Lamport, L., "How to Make a Multiprocesssor Computer That Correctly Executes Multiprocess Programs," *IEEE Transactions on Computersa*, Vol. C-28, IEEE, New York, pp. 690-691 (1979).

[Lenoski and Weber 1995] Lenoski, K. E., and Weber, W. D., Scalable Shared-Memory Multiprocessing, Morgan Kaufmann, San Francisco, CA (1995).

[Lenoski et al. 1992] Lenoski, D., Laudon, J., Gharachorloo, K., Weber, W. D., Gupta, A., Hennessy, J., Horowitz, M., and Lam M. S., "The Stanford Dash Multiprocessor," *IEEE Computer*, Vol. 25, No.3, pp. 63-79 (1992).

[Lenoski et al. 1993] Lenoski, D., Laudon, J., Joe, T., Nakahira, D., Steves, L., Gupta, A., and Hennesssy, J., "The DASH Prototype: Logic Overhead and Performance," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 4, No. 1, pp. 41-61 (1993).

[Li 1986] Li, K., "Shared Virtual Memory on Loosely Coupled Multiprocessors," Ph.D. Dissertation, Technical Report No. YALE/DCS/RR-492, Department of Computer Science, Yale University (September 1986).

[Li 1988] Li, K., "IVY: A Shared Virtual Memory System for Parallel Computing," In: *Proceedings of the International Conference on Parallel Processing*, IEEE New York, pp. 94-101 (August 1988).

[Li and Hudak 1989] Li, K., and Hudak, P., "Memory Coherence in Shared Virtual Memory Systems," *ACM transactions on Computing Systems*, Vol. 7, No. 4, pp. 321-359 (1989).

[Li and Schaefer 1989] Li, K., and Schaefer, R., "A Hypercube Shared Virtual Memory System," In: *Proceedings of the International Conference on Parallel Processing*, Pennsylvania State University Press, pp.125-132 (1989).

[Lilja 1993] Lilja, D. J., "Cache Coherence in Large-Scale Shared-Memory Multiprocessors: Issues and Comparisons," *ACM Computing Surveys*, Vol. 25, pp. 303-338 (1993).

[Lipton and Sandberg 1988] Lipton, R. J., and Sandberg, J. S., "Pram: A Scalable Shared Memory," Technical Report No. CS-TR-180-88, Princeton Unibersity (1988).

[Liskov 1988] Liskov, B., "Distributed Programming in Argus," *Communications of the ACM*, Vol. 31, No. 3, pp. 300-313 (1988).

[Minnich and Faraber 1989] Minnich, R. G., and Faraber, D. J., "The Mether System: A Distributed Shared Memory for SunOS 4.0," In: *Proceedings of the 1989 Summer Usenix Conference*, pp.51-60 (1989).

[Minnich and Farber 1990] Minnich, R. G., and Farber, D. J., "Reducing Host Load, Network Load, and Latency in a Distributed Shared Memory," In: *Proceedings of the 10th International Conference on Distributed Computing Systems*, IEEE, New York (June 1990).

[Mosberager 1993] Mosberager , D., "Memory Consistency Models, "Technical Report No. TR 93/11, Department of Computer Science, Univefrsity of Arizona (1993).

[Nitzberg and Virginia Lo 1991] Nitzberg, N., and Virginia Lo, "Distributed Shared Memory: A Survey of Issues and Algorithms," *IEEE Computer*, Vol. 24, No. 11, pp. 52-60 (1991).

[Oguchi et al. 1995] Oguchi, M., Aida, H., and Saito, T., "A Proposal for a DSM Architecture Suitable for a Widely Distributed Environment and Its Evaluation," In: *Proceedings of the 4th International Symposium on High Performance Distributed Computing*, IEEE, New York (August 1995).

[Ramachandaran and Khalidi 1989] Ramachandaran, U., and Khalidi, M. Y. A., "An Implementation of Distributed Shared Memory," *First Workshop on Experiences with Building Distributed and Multiprocessor Systems*, Usenix Association, Berkeley, CA, pp. 21-38(1989).

[Sane et al. 1990] Sane, A., MacGregor, K., and Campbell, R., "Distorbuted Virtuarl Memory Consistency Protocols: Design and Performance," *Second IEEE Workshop on Experimental Distributed Systems*, IEEE New York, pp. 91-96 (October 1990).

[Scheurich and Dubois 1988] Scheurich, C., and Dubois, M., "DynamicPage Migration in zMultiprocessors with Distributed Global Memory," In:*Proceedings of the 8th International Conference on DistributedComputing Systems*, IEEE Computer Society Press, Los Alamitos, CA, pp.162-169(June 1988).

[Shrivastava et al. 1991] Shrivastava, S., Dixon, G. N., and Parrington, G. D., "An Overview of the Arjuna Distributed Programming System," *IEEE Software*, pp. 66-73 (January 1991).

[Singhal and Shivaratri 1994] Singhal, M., and Shivaratri, N. G., *Advanced Concepts in Operating Systems*, McGraw Hill, New York (1994).

[Sinha et al. 1991] Sinha. P. K., Ashihara, H., Shimizu, K., and Maekawa, M., "Flexible User-Definable Memory Coherence Scheme in Distributed Shared Memory of GALAXY," In: *Proceedings of the 2nd European Distributed Memory Computing Conference (EDMCC2)*, Springer-Verlag, New York, pp.52-61 (April 1991).

[Smith 1982] Smith, A. J., "Cache Memories," *ACM Computing Surveys*, Vol. 14, No. 3, pp. 437-530, New York, NY (1982).

[**Stumm and Zhou 1990**] Stumm, M., and Zhou, S., "Algorithms Implementing Distributed Shared Memory," *IEEE Computer*, Vol. 23, No. 5, New York, NY, pp. 54-64 (1990).

[**Tam and Hsu 1990**] Tam, V., and Hsu, M., "Fast Recovery in Distributed Shared Virtual Memory Systems," In: *Proceedings of the 10th International Conference on Distributed Computing Systems*, IEEE, New York, pp. 38-45 (May-June1990).

[**Tam et al. 1990**] Tam, M. C., Smith, J. M., and Farber, D. J., "A Taxonomy-Based Comparison of Several Distributed Shared Memory Systems," *Operating Systems Review*, Vol. 24, pp. 40-67 (1990).

[**Tartalja and Milutinovic 1996**] Tartalja, I., and Milutinovic, V. (Eds.), *The Cache Coherence Problem in Shared-Memory Multiprocessors: Software Solutions*, IEEE Computer Society Press, Los Alamitos, CA (1996).

[**Theel and Fleisch 1995**] Theel, O. E., and Afleish, B. D., "Design and Analysis of Highly Available and Scalable Coherence Protocols for Distributed Shared Memory Systems Using Stochastic Modeling," In: *Proceedings of the 24th Annual International Conference on Parallel Processing*, IEEE, New York (August 1995).

[**Wu and Fuchs 1989**] Wu, K. L., and Fuchs, W. K., "Recoverable Distributed Shared Virtual Memory: Memory Coherence and Storage Structures," In: *Proceedings of the 19th International Symposium on Fault-Tolerant Computing*, pp.520-527 (June 1989).

[**Yen et al. 1985**] Yen, D. W. L., Yen, W. C., and Fu, K., "Data Coherence Problem in a Multicache System," *IEEE Transactions on Computers*, Vol.C-34, No. 1, pp. 56-65, New York, NY (1985).

[**Zhou et al. 1990**] Zhou, S., Stumm, M., and Mcinerney, T., "Extending Distributed Shared Memory to Heterogeneous Environments," In: *Proceedings of the 10th International Conference on Distributed Computing Systems*, IEEE, New York, pp. 30-37 (May-June 1990).

[**Zhou et al. 1992**] Zhou, S., Stumm, M., Li, K., and Wortman, D., "Heterogeneous Distributed Shared Memory," *IEEE Transactions on Parallel and Distributed Systems*, Vol.13, No.5, New York, NY, pp.540-554 (1992).

インターネット参考文献

分散共有メモリについての参考文献は以下のところで見つけることができる。

ftp:ftp.cs.umanitoba.ca/pub/bibliographies/Parallel/dsm.html

分散メモリシステムについての参考文献は以下のところで見つけることができる。
ftp:ftp.cs.umanitoba.ca/pub/bibliographies/Parallel/distmem.html

キャッシュメモリと関連する話題に関する参考文献は、以下のところで見つけることができる。
ftp:ftp.cs.umanitoba.ca/pub/bibliographies/Misc/cache.htm

単一アドレス空間のオペレーティングシステム (SASOS) と関連する話題に関する参考文献は、以下のところで見つけることができる。

ftp:ftp.cs.umanitoba.ca/pub/bibliographies/Os/sasos.html

第6章

同期

6.1 はじめに

分散システムは、空間的に分離して、並行に動作する別個のプロセスの集まりからなっている。複数の**並行プロセス (concurrent processes)** をもつシステムにおいて、システム資源 (resource)(ハードウェア又はソフトウェア) を並行に動作するプロセス間で共有することは経済的である。このような状況において、共有は協調的 (cooperative) 又は競合的 (competitive) である。すなわち、使用可能な資源の数は限定されているので、あるプロセスが資源を競合するとき、他の並行して動作しているプロセスの行動に必然的に影響を与える。例えば、複数のプロセスから同時に使用できないような資源 (例えば磁気テープ装置など) に対して、他のプロセスがその資源を使用しているならば、それを使用しようとするプロセスは待たなければならない。場合によっては、所望の性能を達成するために、又は実行されている計算上の都合で、並行プロセスは協調しなければならない。プロセス協調の典型例は、お互いに**生産者-消費者 (producer-consumer)** 又は**クライアント-サーバ (client-server)** 関係を分担する 2 つのプロセスを含んでいる。例えば、クライアントプロセスとファイルサーバプロセスはファイルアクセス操作を実行するときには協調しなければならない。協調的又は競合的共有は共に、適正な相互動作の結果を保証するために、動作のルールを固守することを要求する。適正な相互動作を保証するためのルールは**同期 (synchronization)** 機構の形で実装される。本章は分散システムに適した同期機構について述べる。特に、次の同期関連問題を論じる。

■クロック同期 (clock synchronization)
■事象順序制御 (event ordering)
■相互排他 (mutual exclusion)
■デッドロック (dead lock)
■選任アルゴリズム (election algorithms)

6.2　クロック同期

すべてのコンピュータにおいて、現時刻の把握、及び CPU 使用、ディスク I/O 等においてプロセスが使用する時間を計算するような各種の計数目的のためにも、使用するユーザを適切に管理できるように、タイマー機構 (timer mechanism)(**コンピュータクロック (computer clock)** と呼ばれる) が必要である。分散システムにおいて、あるアプリケーションはシステムの複数のノードで並行して動作するかもしれない。正しい結果を得るために、ある種の分散アプリケーションではノードのクロックを相互に同期させることを必要とする。例えば、分散オンライン予約システムが公平であるためには、たった 1 つの残りの座席が異なるノードからほとんど同時に要求された場合、たとえわずかな時間の差であれ、最初に要求を行った依頼人に座席が予約されるべきである。このことを保証することは、もしシステムのノードの間でクロックが同期していないなら、不可能であろう。分散システムにおいて、同期したクロックによって、あるノードに始まり別のノードで終わる分散動作の動作時間を計測することも可能となる。例えば、あるノードから別のノードまでメッセージを転送するのに要する時間の計測が可能となる。この場合、もし送信者と受信者のクロックが同期していないと、正しい結果を得ることは困難である。分散システムにおいて同期クロックの他のアプリケーションが幾つかある。それらのアプリケーションの例は [Liskov 1993] に述べられている。

上記の議論は、分散システムのクロックを適切に同期させるために適切なアルゴリズムを考案して使用することは分散オペレーティングシステム設計者の仕事であることを示している。本節ではそのようなアルゴリズムについて述べる。しかし、これらのアルゴリズムのより良い理解のために、まずコンピュータクロックがどのように実装され、分散システムのクロック同期における主要な問題は何なのかを議論する。

6.2.1 コンピュータクロックの実装

コンピュータクロックは通常、3つの要素からなっている。すなわち、一定の周波数で発振する水晶 (quartz crystal)、**カウンタレジスタ (counter register)** 及び**定数レジスタ (constant register)** である。定数レジスタは、水晶の発振周波数に基づき決定される一定値を格納するのに使用される。カウンタレジスタは水晶の発振を追跡し把握するのに使用される。すなわち、カウンタレジスタの値は水晶の各発振ごとに1ずつ減算される。カウンタレジスタの値がゼロになると、割り込みが発生し、カウンタレジスタの値は定数レジスタの値に再初期化される。各割り込みは**クロックティック (clock tick)** と呼ばれる。

コンピュータクロック機能を日常生活に使用する通常の時計とするために、次のことが必要となる。

1. 定数レジスタの値を1秒間に60クロックティックが発生するように選択する。

2. コンピュータクロックを**実時間 (real time)** (外部クロック) に同期させる。このために、システムにさらに2つの値が格納される。すなわち、固定の開始日付時刻及びティック数である。例えば、UNIXにおいて、1970年1月1日0時0分に時刻が始まる。最初の立ち上げ時に、システムはオペレータに現在日付と時刻を問い合わせてくる。システムは入力された現在日付時刻を固定の開始日付時刻から始まるティック数に変換する。クロックティックのたびに、割り込み処理ルーチンはクロックを進行させ続けるために、ティック数の値を1ずつ加算する。

6.2.2 クロックのドリフト

水晶は一定の周波数で発振するので、クロックは一定の速さで進行する。しかし、水晶の個体差によって、2つのクロックの速さは通常、互いに異なる。最初に2つのクロックを同じ値にどれだけ正確に合わせたとしても、2つのクロックの間での発振周期の相違は微少であるが、多数の発振の間に、その差は累積されて、2つのクロックの時間差として現れてくる。それゆえに、時間の経過にともない、初期設定に使用した実時間クロックからコンピュータクロックはドリフトする。水晶発振器に基づくクロックでは、ドリフト率は約 10^{-6} であり、1,000,000秒、又は11.6日に1秒である [Coulouris et al. 1994]。した

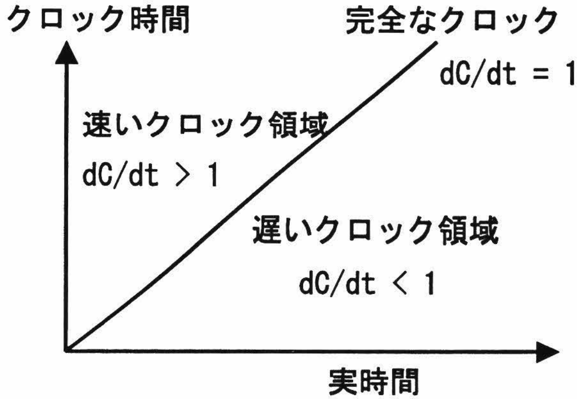


図 6.1　遅い、完全、速いクロック

がって、コンピュータクロックは、正確さを維持するために、実時間クロックに周期的に再同期させなければならない。正常なクロックでさえも必ずしも完全な時間を維持しているわけではない。有限の時間幅において実時間からのドリフトの量に限界があるならば、クロックは正常とみなされる。

より正確には、実時間が t のとき、クロック p の時間の値が $C_p(t)$ であると想定しよう。世界中のすべてのクロックが完全に同期していると、すべてのクロック p と時間 t において $C_p(t) = t$ である。すなわち、もし C があるクロックの時間値を表すなら、理想的には dC/dt が 1 である。それゆえに、もし許容される最大ドリフト率 (maximum drift rate) が ρ ならば、そして次の条件が成り立つならば、クロックは正常であると言われる。

$$1 - \rho \leq dC/dt \leq 1 + \rho$$

図 6.1 に示されるように完全なクロックとの同期の後に、遅い又は速いクロックは完全なクロックから反対の方向にドリフトする。これは遅いクロックは $dC/dt < 1$ であり、速いクロックは $dC/dt > 1$ であることによる。

分散システムは、それぞれが独自の速さの独自のクロックをもつ複数のノードからなっている。すべてのクロックはゼロではないドリフト率をもっているので、ある分散システムのクロックの集合は、何らかの周期的な再同期がなくては、同期がとれた状態を維持することはできない。図 6.1 は、システム全体

にわたって全体の時間基準を維持するためには、分散システムの各ノードはローカルクロックを周期的に再同期させなければならないことを意味する。図6.1から、遅い及び速いクロックは完全クロックから反対の方向にドリフトすることを思い出されたい。したがって、2つのクロックのうち、一つが遅く、一つが速い場合、2つのクロックが同期してから Δt の後に、2つのクロックの時間値間の最大の偏差は $2\rho\Delta t$ となる。したがって、あるクロックの集合において、どの2つのクロックも差が δ よりも大きく異ならないことを保証するために、集合の中の各クロックは、2つの再同期動作間の時間間隔は $\delta/2\rho$ より小さいか等しい間隔で、周期的に再同期しなければならない。したがって、ただ1つのコンピュータクロックが実時間クロックと同期する集中システムとは異なり、分散システムは以下のタイプのクロック同期を必要となる。

1. **実時間 (又は外部) クロックとコンピュータクロックとの同期**:このタイプの同期は、主に実時間アプリケーションのための要求である。すなわち、外部クロックとの同期によって、システムが他のシステム又はユーザとの間で事象のタイミングについての情報を交換することが可能となる。

コンピュータクロックを実時間と同期させるための基準としてよく使用される外部時刻源は UTC(Coordinated Universal Time ：協定世界時) である。UTC は国際標準である。多くの標準化機関が UTC を、ラジオ、電話及び衛星によって配信している。例えば、米国の WWV ラジオ局、及び GEOS(Geostationary Operational Environmental Satellites ：環境静止衛星) はこれらの2つの標準化団体である。市販の装置 (時間プロバイダ (time provider) と呼ばれる) はこれらの信号を受信し解釈することができる。時間プロバイダ装置を保有するコンピュータはそのクロックをこれらのタイミング信号に同期させることができる。

2. **システム内の異なるノードのクロックとの相互 (又は内部) の同期**:このタイプの同期は主に、ある分散システムのすべてのノードにわたって一貫した時間の観点を必要としたり、1つのノードに始まり別のノードに終わる分散動作の経過時間を測定が必要なアプリケーションにおいて必要である。

外部同期クロックは内部同期もなされていることに注意されたい。しかし、時間の経過とともに内部同期クロックは外部時間からドリフトするので、逆は真ではない。

6.2.3 クロック同期問題

2つのクロックを完全に同期させることはできないことを述べた。それゆえ、実際には、特定の時間において2つのクロックの時間差がある規定の定数δよりも小さいならば、2つのクロックは同期していると言える。2つのクロックの時間差は**クロックスキュー (clock skew)** と呼ばれる。それゆえ、どのクロックスキューもδよりも小さいならば、クロックの集合は同期していると言える。

クロック同期 (clock synchronization) は各ノードが互いにクロック値を読み取ることを必要とする。他のクロックを読み取るための実際の機構はアルゴリズムごとに異なる。しかし、実際の読み取り機構がどのようなものであれ、システムにおける他ノードのクロックスキューについては近似的な観測を得ることができるのみである。誤差は主に、クロック信号を配信、又は1つのノードから他のノードへのクロックメッセージを配信するために使用されるメッセージパッシング動作中の予測不可能な通信遅れによって生じる。2つのノード間の予測不可能な通信遅れの最小値は、転送誤りがなく他のシステム負荷がない場合の空メッセージを準備し、転送し、そして受信するのに要する時間を計数することにより計算できる。しかし、一般に、通信遅れ時間の上限値を計算することは不可能である。その理由は、通信時間遅れは、システム内で並行に行われる通信と計算の量、転送誤りによりメッセージの複数回転送、及びページフォールト、プロセス切り替え、通信ルートの変更などの他のランダム事象に依存するからである。

クロック同期における重要な事項は、時間は決して後退してはならないという点である。その理由は、場合によって危険な結果をもたらしかねない操作の重複のような重大な問題を引き起こすかもしれないからである。同期のときに、速い (進んでいる) クロックは遅れさせなければならないことに注意されたい。しかし、もし速いクロックの時間が瞬間的に実際の時間に再調整されると、そのクロックは時間を後退することになる。それゆえに、クロック同期アルゴリズムは通常、瞬間的に正しい時間に再調整する代わりに、速いクロックに緩やかな変化 (遅れ) を導入するように設計される。これを行う1つの方法は割り込みルーチンに工夫をこらすことである。クロック同期アルゴリズムから割り込みルーチンがクロックを遅らすように指示を受けると、各割り込みについてクロックに加算する時間量を再調整する。例えば、通常状態時に各割り込みに8ミリ秒加算されているならば、減速時には、訂正が完了するまでの間、各割り込みに7ミリ秒のみ加算する。必ずでもないが、円滑な再調整のため

に、もし遅れていると検出すると、瞬間的に正しい時間に再調整するのではなく、各割り込みに 9 ミリ秒加算することにより、割り込みルーチンはクロックを速めることもある。

6.2.4 クロック同期アルゴリズム

クロック同期アルゴリズムは集中型 (centralized) と分散型 (distributed) に大別される。

集中型アルゴリズム

集中型のクロック同期アルゴリズムにおいて、1 つのノードが実時間受信機を持っている。このノードは通常、**時間サーバノード (time server node)** と呼ばれ、このノードのクロック時間は正しいとみなされ基準時間 (reference time) として使用される。アルゴリズムの目標は、他のすべてのノードのクロックを時間サーバのクロック時間に同期させることである。時間サーバノードの役割によって、集中型クロック同期アルゴリズムは 2 つのタイプに分類される。すなわち、**受動型 (passive)** 時間サーバと**能動型 (active)** 時間サーバである。

受動型時間サーバの集中型アルゴリズム

この方法において、各ノードは周期的 ($\sigma/2 \cdot \rho$ よりも小さいか等しい周期の間隔で) に時間サーバにメッセージ ("time = ?") を送信する。時間サーバがメッセージを受信すると、メッセージ ("time = T") をできるだけすばやく応答する。ここで T は時間サーバノードのクロックにおける現在時間である。クライアントノードが "time = ?" メッセージを送信したとき、クロック時間が T_0 であり、"time = T" メッセージを受信したとき、クロック時間が T_1 であったとする。T_0 及び T_1 は、クライアントノードの同じクロックでの計測値であるので、他に情報がない状況では、時間サーバからクライアントへの "time = T" メッセージの伝搬に要する最良の推定は $(T_1 - T_0)/2$ である。それゆえ、クライアントノードで応答が受信されるとき、そのクロックは $T + (T_1 - T_0)/2$ に再調整される。

2 つのノードの間でメッセージ伝搬時間に予測不可能な変動があるので、クライアントのノードクロックの現在時間を計算するために T に加算される時間として $(T_1 - T_0)/2$ は必ずしも非常に良い推定というわけではない。この推定値を改善するための幾つかの提案がなされている。そのような 2 つの方法を以

下に述べる。最初の方法は、ある追加情報が利用可能であることを仮定し、第 2 の方法は追加情報がないことを仮定する。

1. この方法では、時間サーバが割り込みを処理し、"time = ?" 要求メッセージを処理する概略の時間が既知であることを仮定している。この時間が I であるとする。そうすると、"time = T" メッセージが時間サーバからクライアントノードに伝搬するのに要する時間のより良い推定は $(T_1 - T_0 - I)/2$ である。それゆえ、この方法では、クライアントノードに応答が受信されるとき、そのクロックは $T + (T_1 - T_0 - I)/2$ と再調整される。

2. この方法は Cristian [1989] によって提案された。この方法において、幾つかの $T_1 - T_0$ の測定を行い、これらの測定において、ある閾値 (threshold) を越えるものは信頼できないとして棄却する。残った測定値の平均値が計算され、その 2 分の 1 が T に加算されるべき値として使用される。代替の方法として、$T_1 - T_0$ が最小である計測値は最も正確な値とみなされ、この値が T に加算される値として使用される。この方法の 1 つの限界は、T に加算される値を推定するための測定の数を制限する必要があるという点である。その理由はこれらの測定はメッセージトラフィックの発生及びアルゴリズムによるオーバーヘッドに直接関係するからである。

能動的時間サーバ集中型アルゴリズム

受動的時間サーバの方法において、時間サーバは他のノードからの時間要求に対して単に応答するのみである。他方、能動的時間サーバの方法では、時間サーバは周期的にクロック時間 ("time = T") をブロードキャストする。他のノードはブロードキャストメッセージを受信し、メッセージの中のクロック時間を自己のクロックを訂正するのに使用する。各ノードは、メッセージ "time = T" が時間サーバから自己のノードに伝搬するのに必要な概略の時間 T_a について事前の知識をもっている。それゆえ、ブロードキャストメッセージがあるノードに到着すると、そのノードのクロックは時間 $T + T_a$ に再調整される。この方法の主な欠点はフォールトトレラントではないという点である。ある通信障害によりブロードキャストメッセージがあるノードに過剰に遅れて到着すると、そのノードのクロックは間違った値に再調整されるであろう。この方法の別の欠点は、ネットワークによってブロードキャスト機能がサポートされることが必要である点である。

上記の欠点を克服する別の能動的時間サーバアルゴリズムは **Berkeley アルゴリズム (Berkeley algorithm)** である。これは Berkeley UNIX で動作するコンピュータのグループの内部同期のために Gusella and Zatti [1989] によって提案された。このアルゴリズムにおいて、時間サーバは周期的にメッセージ("time = ?") をグループ内のすべてのコンピュータに送信する。このメッセージを受信すると、各コンピュータは自己のクロック値を時間サーバに返送する。時間サーバは各ノードから自己のノードへメッセージが伝搬する概略時間の値を事前に知っている。この知識に基づき、最初に、時間サーバは応答メッセージのクロック値を再調整する。それから、すべてのコンピュータ (自己も含めて) のクロック値をフォールトトレラント的平均を行う。フォールトトレラント的平均を行うために、時間サーバは、互いにある規定された値より大きく異ならないクロック値のサブ集合を選択し、このサブ集合内のクロック値のみで平均を行う。この方法は、通常の平均を行うと有害な結果をもたらすようなクロック値をもつ信頼できないクロックからの読み取り値を除去する。

計算された平均値はすべてのクロックが再調整されるべき現時刻である。時間サーバは自己のクロックをこの値に再調整する。しかし、計算された時間を他のコンピュータに送信する代わりに、時間サーバは各個別のコンピュータのクロックが再調整を必要とする値を送信する。この値は正又は負の値であり、各ノードから時間サーバへのメッセージ伝搬に要する概略時間について時間サーバがもつ知識に基づき計算される。

集中型クロック同期アルゴリズムは 2 つの主な欠点をもっている。

1. **単一障害 (single-point failure)** 依存である。もし時間サーバノードが故障すると、クロック同期動作が行われなくなる。これがシステムを不信頼にしている。理想的には、分散システムは個別のノードよりも、高信頼であるべきである。もし一つがダウンしても残りが機能を正しく継続するべきである。

2. **規模拡張性 (scalability)** の観点から、すべての時間要求が単一の時間サーバによって処理されるということは、一般に受け入れ難い。大規模なシステムでは、このような手法は 1 つのプロセスに過大な負荷を与えることになる。

分散型アルゴリズムはこれらの欠点を克服する。

分散型アルゴリズム

外部同期されているクロックは内部同期もされていることを思い出された

い。すなわち、もし各ノードのクロックが独立に実時間と同期しているとすると、システムのすべてのクロックは互いに同期していることとなる。それゆえ、クロック同期の単純な方法は、システムの各ノードに実時間受信機を備え、各ノードが独立に実時間に同期することである。複数の実時間クロック (各ノードごとに 1 つ) は通常この目的に使用される。

理論的には、この方法では、クロックの内部同期は必要ではない。しかし、実際には、実時間クロックの本質的な不正確さにより、異なる実時間クロックは異なる時間を生成する。それゆえ、よりよい精度のために内部同期が通常行われる。次の 2 つの方法の 1 つがこの場合の内部同期に用いられる。

全体平均分散型アルゴリズム

この方法では、各ノードのクロックプロセスは、自己の時間がある整数 i に対し $T_0 + iR$ に等しいとき、自己のクロック時間を特別の「再同期 (resync)」メッセージの形式でブロードキャストする。ここで、T_0 は過去にすべてのノードで合意された固定の時間であり、R はシステムに含まれるノードの総数、最大許容ドリフト率、その他に依存するシステムパラメータである。すなわち、再同期 メッセージは、固定長の再同期間隔の始まりごとに、各ノードからブロードキャストされる。しかし、異なるノードのクロックはわずかな違いの率で動作しているので、これらのブロードキャストはすべてのノードから同時に起きることはないであろう。

クロック値をブロードキャストした後に、あるノードのクロックプロセスは時間 T だけ待つ。ここで T はアルゴリズムによって決定されるパラメータである。この待っている間に、クロックプロセスは他のノードによってブロードキャストされた再同期メッセージを集める。各再同期メッセージに対して、クロックプロセスは、自己のクロックに従って、メッセージを受信した時間を記録する。待ち期間の終わりに、クロックプロセスは、再同期メッセージを受信した時間に基づき、自己のクロックの他の各ノードとのスキューを推定する。それから、推定スキューのフォールトトレラント平均を計算し、次の再同期周期の開始時点までに自己のクロックを訂正するのにその値を使用する。

全体平均アルゴリズムは推定スキュー値のフォールトトレラント平均を計算する方法によって異なる。2 つのよく使用されるアルゴリズムを以下に述べる。

1. 最も単純なアルゴリズムは推定スキュー値を平均し、自己のクロックの訂正に使用することである。しかし、不正なクロックが平均値に与える影響を

制限するため、各ノードに関する推定スキュー値がある閾値に対して比較され、閾値より大きなスキューは推定スキュー値の平均計算の前に、ゼロに設定される。

2. 別のアルゴリズムでは、各ノードは不正クロックの影響を制限するために、まず m 個最大、及び m 個最小の推定スキューを除去し、次に残りのスキューから平均を計算し、それから自己のクロックの訂正に用いる。m の値は通常、クロック (ノード) の総数に基づき決定される。

個別平均分散アルゴリズム

全体平均アルゴリズムは規模拡張性に問題がある。その理由はネットワークにブロードキャスト機能が必要であり、大量のメッセージトラフィックが生成されるからである。それゆえ、このアルゴリズムは小規模ネットワーク、特に完全接続トポロジー (各ノードが互いに直接の通信リンクをもつ) をもつネットワークに適している。個別平均アルゴリズムは、全体平均アルゴリズムのこれらの欠点を克服することを試みている。この方法では、分散システムのノードはある種のパターン、例えばリング又はグリッドで論理的にならべられている。周期的に、各ノードはそのクロック時間をリング、グリッド、又はその他の構造における隣接ノードと交換し、そして自己のクロック時間と隣接のクロック時間との平均値に自己のクロック時間を設定する。

6.2.5 事例研究：分散時間サービス

クロックを同期させ、広範囲な形態で相互接続されたネットワークにわたって時間情報を提供するための2つの一般的なサービスは分散時間サービス (Distributed Time Service：**DTS**) とネットワーク時間プロトコル (Network Time Protocol：**NTP**) である。DTS は **DCE(Distributed Computing Environment：分散計算環境)** の1つの構成要素であり、DCE で動作しているコンピュータのネットワークのクロックを同期させるために使用される。NTP はインターネットにおけるクロック同期に使用される。DTS をクロック同期の事例研究として以下に概説する。NTP の詳細は [Mills 1991] に述べられている。

DCE システムにおいて、各ノードは **DTS クライアント (DTS client)** 又は **DTS サーバ (DTS server)** として構成される。各 DTS クライアントノードにおいて、**DTS クラーク (DTS clerk)** と呼ばれるデーモンプロセスが動

いている。自己のクロックを同期させるため、各DTSクラークは同一LAN上のDTSサーバに時間情報を求める要求を行う。DTSサーバは要求に基づき、DTSクラーク又は他のDTSサーバに時間情報を供給する。各DTSサーバはその存在を公告するために、名前をLANプロファイルにイクスポートする。

DTSは時間を単一の値として定義しない。その代わり、時間は正しい時間を含む間隔として表現される。値の代わりに、間隔を用いることにより、DTSはユーザに基準時間からどれだけ離れているかという明確な情報を提供する。

DTSクラークは自己のクロックを次の方法で同期させる。自己のクロックのドリフト率を監視し、自己のクロックが許容範囲を越えたことを発見すると、LAN上のすべてのDTSサーバにRPCを用いて時間を要求することにより再同期を開始する。このメッセージを受信すると、各DTSサーバはサーバ自身のクロックに基づく時間間隔を含む回答を返す。受信した回答から、DTSクラークは次の方法(例えば図6.2)により新しい時間値を計算する。まず、過半数の時間間隔と重複しない時間間隔は不正とみなされ、廃棄される。例えば、図6.2において、DTSサーバ3によって供給される値は廃棄される。それから、残りの間隔の最大重複間隔が計算される。DTSクラークは、この間隔の中点にクロック値を再設定する。しかし、一度にクロックを計算値に再設定する代わりに、クロック時間における変化を緩やかにするために、知的割り込みルーチンが使用される。

新しいクロック値を計算するため重複法を用いることは、DCEシステムにおいて各LAN上に時間情報を提供する少なくとも3つのDTSサーバが存在することが推奨される。

DTSクラークがそれぞれのクロックをDTSサーバに同期させるのに加えて、LAN上のDTSサーバもまたサーバ同士で周期的に通信を行い、それぞれのクロックについて相互に同期を維持する。DTSサーバも、新しいクロック値を計算するために、図6.2のアルゴリズムを使用する。

いままで、同じLANに所属するノードのクロックをどのようにして同期させるかについて述べた。しかし、DCEシステムは幾つかの相互接続されたLANをもつこともある。この場合に、ネットワーク内のすべてのノードのクロックを同期させる必要が生じる。このため、各LANの1つのDTSサーバが全体サーバ(global server)と指定される。外部同期は必ずしも必要ではないが、全体サーバはUTC信号を受信するための時間プロバイダ装置を備える。すべてのLANの全体サーバは周期的に通信を行い、相互にそれぞれのクロックの同期を維持する。あるLANにおける全体サーバはまたDTSサーバである

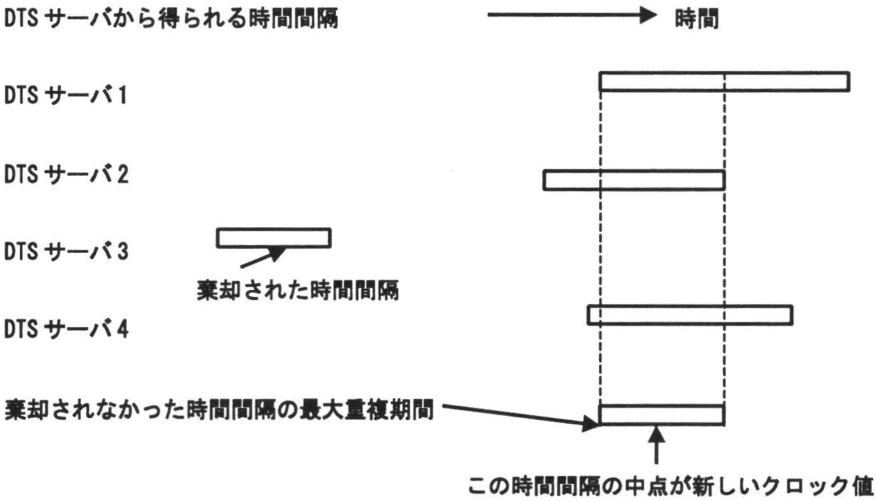


図 **6.2** DTS において得られた時間間隔から新しいクロック値の計算

ので、そのクロック値は自動的に、LAN 上の他のノードのクロックを同期させるのに使用される。このようにして、DTS はネットワークにおけるすべてのノードのクロックを同期させる。

ユーザは DTS を直接アクセスことができないという意味で DTS は DCE ユーザにとって透過的である。しかし、DTS アプリケーションプログラミングインタフェース (**application programming interface : API)** は、DTS アプリケーションが、それらの実行を制御するための時間に関係する処理を行うことができる豊富な手続きライブラリーを提供する。特に、現在時間の獲得、時間の 2 進数と ASCII 表現の変換、2 進数時間情報の操作、2 つの時間の比較、時間の算術演算、時間ゾーンの獲得、その他のための手続きライブラリーがある。加えて、DTS に対する管理的インタフェースがあり、システム管理者が次のような管理的操作を行うことができる。多数の LAN 上の DTS クライアント及び DTS サーバの構成及び動的再構成、LAN の全体サーバが故障の場合に、ある DTS サーバを全体サーバへの変更、どれほどの頻度で再同期をさせるかを決定するための最大許容誤差の設定などの管理的操作である。

6.3 事象の順序

分散システムにおいて、クロックの同期を 5 又は 10 ミリ秒以内に維持することはコストの高い、かなりな仕事である。Lamport [1978] はほとんどのアプリケーションに対し分散システムにおいてクロックを同期させることは必要ではないと観察した。むしろ、分散システムにおいて生起するすべての事象がある方法で全体として順序づけられていることを確かにすることの方が重要である。

事象の**半順序 (partial ordering)** を定めるため、Lamport は**先行生起 (happened-before)** と呼ばれる新しい関係を定義し、先行生起関係に基づく事象の順序つけのための論理クロックの概念を導入した。それから半順序の概念を分散システムにおけるすべての事象の一貫した全体順序へと拡張する分散アルゴリズムを考案した。以下にそれを述べる。

6.3.1 先行生起関係

一連の事象の先行生起関係 (→で表記される) は次の条件を満たす。

- 1. 同じプロセスにおいて、a と b が事象であり、a が b の前に生起するならば、a → b である。

- 2. もし a が 1 つのプロセスによってメッセージを送信する事象であり、b が別のプロセスによって同じメッセージを受信する事象であるならば、a → b である。この条件は、受信者は送信者がメッセージを送信するまでには受信することができないし、送信者から受信者へメッセージを伝搬する時間は常に正の値であるので、因果律 (law of causality) によって成り立つ。

- 3. もし a → b 及び b → c であるならば、a → c である。すなわち、先行生起関係は推移関係 (transitive relation) である

物理的な意味のあるシステムにおいて、ある事象はそれ自体先行して生起することはない、すなわち、どの事象 a についても a → a は真ではない。これは、 先行生起関係はシステムにおけるすべての事象の集合は非反射半順序 (irreflexive partial ordering) であることを意味する。

先行生起関係において、2 つの事象 a 及び b は、 先行生起関係によって関係していないならば、並行 (concurrent) であると言われる。すなわち、 a → b も真ではなく b → a も真ではない場合である。このような状況はもし 2 つの事象

が、メッセージ交換を直接的にも他のプロセス経由で間接的にも行わない異なるプロセスで生起する場合に起きうる。この**並行性 (concurrency)** の定義は2つの事象が生起する時間、又はどちらが先に生起したかについて何も言えないことを単に意味するだけである。すなわち、2つの事象が因果的に互いに他に影響を与えないならば、並行である。この理由によって、 先行生起関係は時に因果的順序 (causal ordering) の関係として知られている。

空間-時間図 (図 6.3 に示すような) は 先行生起関係及び並行事象の概念を示すために、よく使用される。この図において、各垂直線はプロセスを、垂直線上の各点は対応するプロセスにおける事象を、及び各波線は1つのプロセスから他のプロセスへの矢印方向へのメッセージの転送を表す。

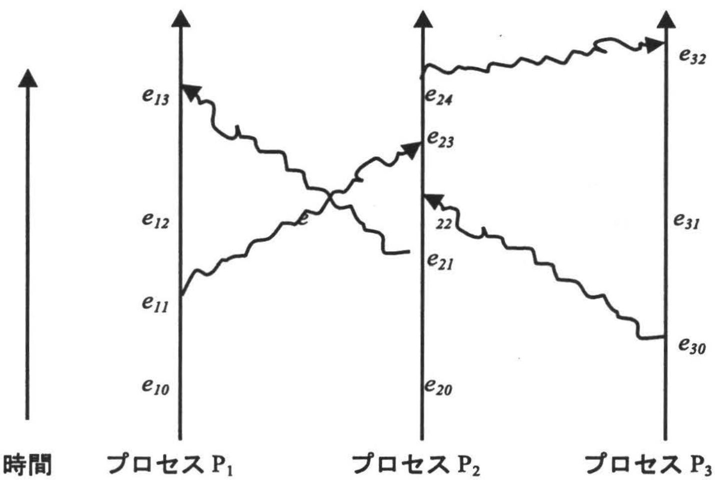


図 **6.3** 3つのプロセスの空間-時間図

この空間-時間図から、2つの事象 a 及び b において、プロセスの時間の進む方向に沿って、及び矢印の方向にメッセージの線に沿って、a から b へのパスが存在するならば、またその場合のみ、a → b が真であることを理解することは容易である。例えば、 先行生起関係によって関係づけられる図 6.3 の幾つかの事象を次に示す。

$e_{10} \rightarrow e_{11} e_{20} \rightarrow e_{24} e_{11} \rightarrow e_{23} e_{21} \rightarrow e_{13}$

$e_{30} \rightarrow e_{24}$(理由 $e_{30} \rightarrow e_{22}$ 及び $e_{22} \rightarrow e_{24}$)

$e_{11} \rightarrow e_{32}$(理由 $e_{11} \rightarrow e_{23}, e_{23} \rightarrow e_{24}$ 及び $e_{24} \rightarrow e_{32}$)

一方、2 つの事象 a 及び b は、a から b 又は b から a へのパスが存在しないなら、その場合に限り並行である。例えば、図 6.3 の並行事象を次に示す。

e_{12}及び $e_{20}e_{21}$及び $e_{30}e_{10}$及び $e_{30}e_{11}$及び $e_{31}e_{12}$及び $e_{32}e_{13}$及び e_{22}

6.3.2　論理的クロックの概念

ある事象 a が事象 b の前に生起したということを決定するために、1 つの共通クロック又は完全に同期したクロックの集合が必要である。分散システムにおいて、これらはどれも不可能であることを既に述べた。それゆえ、分散システムにおいて先行生起関係は、全体に同期した物理クロックを使用することなく定義されなければならない。Lamport [1978] は**論理クロック (logical clock)** の概念を導入することにより、この問題に 1 つの解を与えた。

論理クロックの概念は**タイムスタンプ** (どのクロック時間とも独立な単なる数であるかもしれない) を各システム事象と関連付ける方法である。この関連付けにより、 先行生起関係によって互いに関連付けられた事象は順序付けがなされる。この概念により、各プロセス P_i はクロック C_i をもち、そのプロセスにおけるどの事象にもある数 $C_i(a)$ をアサインする。各プロセスのクロックは、数 $C_i(a)$ の関係について物理時間と何等の関係も仮定していないので論理クロックと呼ばれる。実際に、論理クロックは、実際の計時機構によらず、カウンタによって実現されている。各プロセスが各自のクロックをもつことにより、クロックの全体システムは関数 C で表現され、事象 b に数 $C(b)$ が割り付けられる。ここでもし b がプロセス P_j における事象ならば $C(b) = C_j(b)$ である。

これらのクロックを用いて、先行生起関係によって関連付けられたシステムの事象が適切に順序付けられるならば、システムの論理クロックは正しいとみなされる。それゆえに、システムによって事象に割り付けられた論理クロックのタイムスタンプは次のクロック条件 (clock condition) を満たさなければならない。

どの 2 つの事象 a 及び b に対しても、a → b ならば、$C(a) < C(b)$ である。

逆の条件は、どの 2 つの並行事象も同時に発生する必要があることを意味し、それはすべての並行事象に対して必要以上の制約となるので、逆の条件が成立することを期待できないことに注意されたい。

6.3.3 論理クロックの実装

先行生起関係の定義から、次の条件を満たすならば、上記のクロック条件は満たされる。

C1: もし事象 a 及び b が同じプロセス P_i における事象であり、a が b よりも前に生起するならば、$C_i(a) < C_i(b)$ である。

C2: もし a がプロセス P_i によりメッセージを送信し、b がプロセス P_j でメッセージを受信するならば、$C_i(a) < C_j(b)$ である。

クロック条件を満たすべきこれらの条件に加えて、次の条件がシステムが正常に動作するために必要である。

C3: プロセス P_i と関係するクロック C_i は常に前進し、後退してはならない。すなわち、論理クロックの修正は常にクロックに正の値を加算することによってなされねばならない。決して減算を行ってはならない。

明らかに、論理クロックを実装するどのようなアルゴリズムもこれらの 3 つの条件を満たさねばならない。Lamport によって提案されたアルゴリズムを以下に示す。

C_1、C_2 及び C_3 の条件を満たすために、 **Lamport** のアルゴリズムは次の実装ルールを用いる。

IR_1：各プロセス P_i は事象発生ごとに C_i を増加させる。

IR_2：もし事象 a がプロセス P_i によってメッセージ m を送信する事象ならば、メッセージがタイムスタンプ $T_m = C_i(a)$ を含み、メッセージ m を受信した P_j は現在のクロック値以上又は T_m より大きい値を C_j に設定する。

ルール IR_1 は条件 C_1 が満たされることを保証し、ルール IR_2 は条件 C_2 が満たされることを保証する IR_1 及び IR_2 の両者は条件 C_3 が満たされることを保証する。したがって、単純な実装ルール IR_1 及び IR_2 は論理クロックの正しいシステムを保証する。

論理クロックの実装は例を用いて、よく説明できる。論理クロックのシステムが実際の計時機構を使用せずにカウンタを使用することにより、又は物理クロックを使用することなく実装されることを以下に示す。

カウンタを使用した論理クロックの実装

図 6.4 に示されるように、2 つのプロセス P_1 及び P_2 はそれぞれカウンタ C_1 及び C_2 をもっている。カウンタは論理クロックとして動作する。最初は、カウンタはゼロに初期化され、プロセスに事象を生じるごとにカウンタを 1 増加

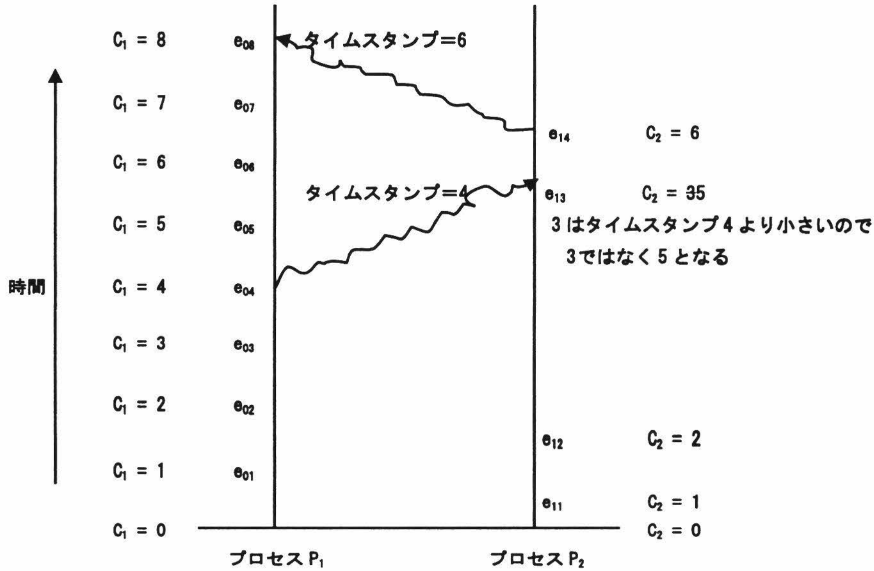


図 6.4　カウンタを使用した論理クロックの実装を示す例

させる。もし事象がメッセージを送信する事象 (例えば事象 e_{04} 及び e_{14}) ならば、プロセスはメッセージの中に増加されたカウンタの値を含ませる。他方、もし事象がメッセージを受信する事象 (例えば事象 e_{13} 及び e_{08}) ならば、カウンタをまず 1 増加させ、次に、増加した値が受信メッセージ中のタイムスタンプより小さいかどうか検査する。もし小さいならば、受信メッセージ (例えば e_{13}) の中のタイムスタンプに 1 加算した値にカウンタを訂正する。もしそうでなければ、カウンタの値はそのままにされる (例えば、e_{08})。

物理クロックを使用した論理クロックの実装

図 6.4 に示したカウンタを用いる方式に代わり、図 6.5 に物理クロックを用いる実装例を示す。この場合、各プロセスはそれぞれに物理クロックを持っている。各クロックは一定の率で動作している。しかし、異なるクロックの率は同じではない。例えば、図 6.5 の例において、プロセス P_1 は 10 回チック (tick) する (時を刻む) 間に、プロセス P_2 のクロックは 8 回のみチックする。

条件 C1 を満たすために、ただ 1 つの要求は、同じプロセスにおける 2 つの事象の間に少なくとも 1 回のチックを行うことである。コンピュータクロック

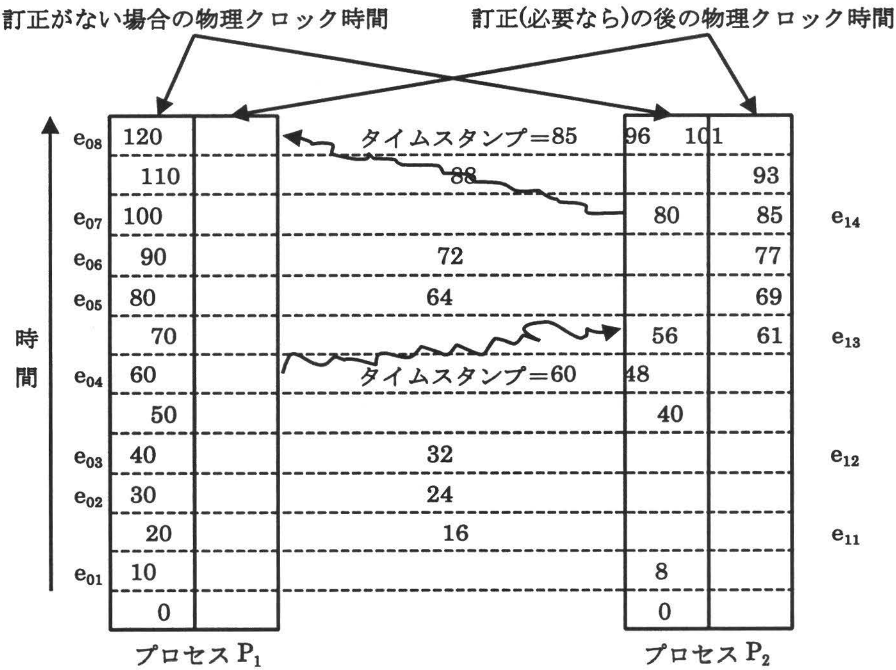


図 6.5　物理クロックを使用した論理クロックの実装を示す例

は近接する 2 つの事象の間でも数回のチックを行うように設計されているので、これは通常問題ではない。条件 C2 を満たすために、メッセージ送信事象 (例えば、事象 e_{04} 及び e_{14}) に対して、メッセージを送信するプロセスはメッセージの中に現在の物理時間を含ませる。そして、メッセージ受信事象 (例えば、事象 e_{13} 及び e_{08}) に対して、受信者のクロックの現在時間がメッセージの中に含まれる時間より小さいか等しいかどうかの検査がなされる。もしそうであるならば、受信者の物理クロックはメッセージに含まれる時間よりも 1 多くクロックを進めるように訂正される (例えば事象 e_{13})。そうでなければ、受信者のクロックはそのままにされる (例えば、事象 e_{08} におけるように)。

6.3.4　事象の全順序

事象間で先行生起関係に基づきシステムにおける事象を順序つけるためにクロック条件を満たすクロックシステムがどのように使用されているかを述べて

きた。事象を生起した時間によって順序つけることが単に必要なだけである。しかし、先行生起関係はシステムにおけるすべての事象集合に**半順序 (partial ordering)** を定めるのみであることに注意されたい。この事象順序付け方式では、先行生起関係 (直接的であっても又は間接的であっても) によって関係付けられていない 2 つの事象 a 及び b はそれぞれに同じタイムスタンプをもつこともあり得る。例えば、もし事象 a 及び b がそれぞれプロセス P_1 及び P_2 において発生し、両者のプロセスのクロックが同じ時間 (例えば 100) であるならば、両者の事象は 100 のタイムスタンプをもつであろう。この状況で、2 つの事象の順序についても何もいえない。それゆえ、システムのすべての事象集合の**全順序 (total ordering)** を付けるために、付加的な要件が必要である。その要件とは、どの 2 つの事象も正確に同じ時間には生起しないということである。この要件を満たすために、 Lamport はプロセスの任意の全順序つけ方法を使用することを提案した。例えば、**プロセス特定番号 (process identity number)** が事象の全順序をつけるために使用されることもある。例えば、上記の状況において、事象 a 及び事象 b のタイムスタンプがそれぞれ 100.001 及び 100.002 となる。ここで、プロセス P_1 及び P_2 のプロセス特定番号がそれぞれ 001 及び 002 である。この方法により、システムにおけるすべての事象の全順序を定めるために、分散システムにおける各事象にユニークなタイムスタンプを割り当てることができる。

6.4　相互排他

あるシステムにおいて、プログラムの動作が正常ならば、複数のプロセスによって同時に使用されてはならない幾つかの資源が存在する。例えば、ファイルは複数のプロセスによって同時に更新されてはならない。同様に、磁気テープ装置やプリンタのような記録用の周辺機器の使用も同時には単一のプロセスに限定されねばならない。それゆえに、このような共有資源に対する排他的アクセスが保証されなければならない。このアクセスの排他性はプロセス間の**相互排他 (mutual exclusion)** と呼ばれる。共有資源に排他的にアクセスする必要のあるプログラムのセクション（領域）は**臨界領域 (critical section)** と言われる。相互排他のために、プロセスが関連する臨界領域を同時に実行することを防止するための手段が導入される。

相互排他を実装するためのアルゴリズムは次の要件を満たさなければならない。

1. **相互排他**: 複数の平行プロセスによってアクセスされる共有資源が存在するとき、いかなる時にもただ1つのプロセスが資源をアクセスするべきである。すなわち、その資源の使用を許可されたプロセスは他のプロセスに使用を許可する前に、資源を開放しなければならない。

2. **飢餓無し (no starvation)**: もし資源を確保していた各プロセスが資源を開放するならば、その結果、各要求は実質的に許可されなければならない。

単一プロセッサシステムにおいて、相互排他はセマフォや、モニタや同様の仕組みを用いて実現される。分散システムにおいて相互排他を実装するための異なるアルゴリズムによって使用される3つの基本的方法を次に述べる。この話題についてさらに学習したい興味ある読者は、[Agarwal 及び Abbadi 1991, Bulgannawar 及び Vaidya 1995, Raynal 1991, Sanders 1987, Suzuki 及び Kasami 1985] を参照されたい。議論を簡潔にするために、各プロセスは異なるノードに存在するものと仮定している。

6.4.1 集中型の方法

集中型の方法では、システムにおけるプロセスの一つがコーディネータ **(coordinator)** として選ばれ (コーディネータ選任のアルゴリズムはこの章の後半に述べる)、臨界領域へ入ることを調整する。臨界領域へ入ろうとする各プロセスは、最初にコーディネータから許可を得なければならない。もし他のプロセスが現在その臨界領域にいないならば、コーディネータは直ちに要求したプロセスに許可を与えることができる。しかし、もし2つ以上のプロセスが同じ臨界領域に入るための許可を求めているならば、コーディネータはあるスケジューリングアルゴリズムにしたがって同時にはただ1つのプロセスにのみ許可を与える。臨界領域を実行した後に、プロセスが臨界領域を出るとき、コーディネータに知らせる。コーディネータは、同じ臨界領域に入るための許可を求めた別のプロセス (もしあれば) に許可を与えることができる。

集中型の方法が使用する相互排他のためのアルゴリズムを以下に例とともに述べる。図6.6に示されるように、システムに1つのコーディネータプロセス (P_c)、3つのプロセス P_1、P_2、P_3 があると仮定する。また、要求はコーディネータが管理する要求待ち行列における**先着優先順序 (first-come, first-served order)** で許可が与えられるものと仮定する。P_1 が臨界領域に入

ることを求めて P_c に要求 (request) メッセージを送信するとする。要求メッセージを受信し、P_c は臨界領域に他のプロセスがいるかどうか検査する。臨界領域に他のプロセスがいないと、P_c は直ちに P_1 に許可を与える応答 (reply) メッセージを返信する。応答が到着すると、P_1 は臨界領域に入る。

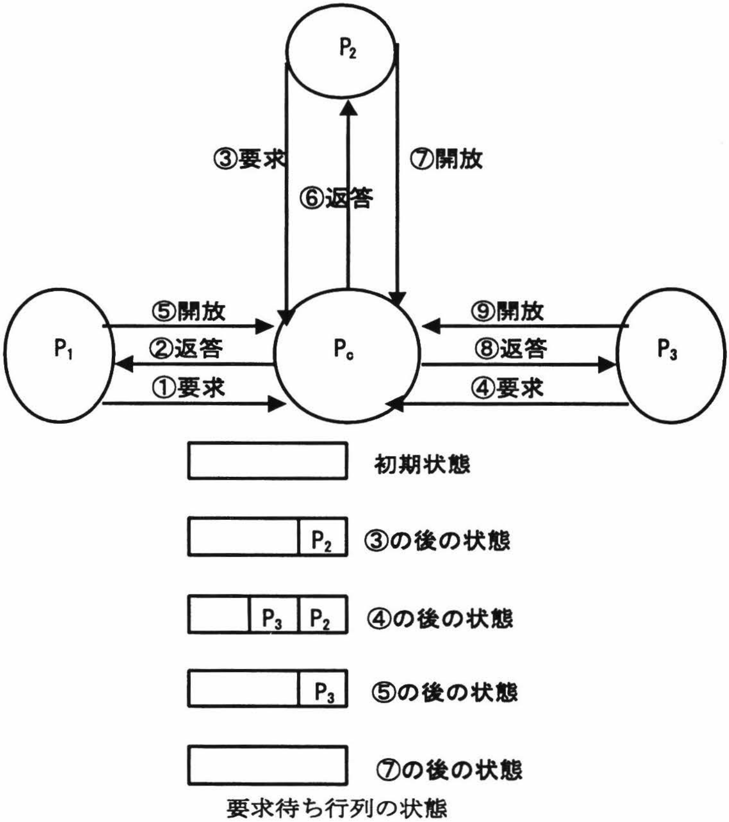


図 **6.6**　相互排他のための集中型方法を示す例

P_1 が臨界領域にいる間に、P_2 が同じ臨界領域に入るための許可を求めて P_c に要求メッセージを送信するとする。P_1 が既に臨界領域にいるので、P_2 は許可を得られない。許可を却下する正確な方法はアルゴリズムによって異なる。ここでは、コーディネータは何も応答せず、要求を行ったプロセスはコーディネータから応答があるまではブロックされたままであると仮定する。それゆえ、P_c は P_2 に直ちに応答を行わずに、その要求を要求待ち行列に入れる。

また、P_1 が臨界領域にいる間に、P_3 も同じ臨界領域に入るための許可を求めるメッセージを P_c に送信するとする。明らかに、P_3 は許可を得られず、P_c から P_3 に応答が直ちに送信されることはない。そしてその要求は要求待ち行列において待たされる。

次に、P_1 が臨界領域を出て、臨界領域の排他的アクセスを解放する解放 (release) メッセージを P_c に送信するとする。解放メッセージを受信すると、P_c は要求待ち行列から最初の要求を取り出し、対応するプロセスに、臨界領域に入るための許可を与える応答メッセージを送信する。この場合、P_c は P_2 に応答メッセージを送信する。

応答メッセージを受信すると、P_2 は臨界領域に入り、臨界領域を出るとき、解放メッセージを P_c に送信する。再び P_c は要求待ち行列から最初の要求 (この場合 P_3 の要求) を取り出し、対応するプロセス (P_3) に応答メッセージを送信する。応答メッセージを受信すると、P_3 は臨界領域に入り、臨界領域を出るとき、解放メッセージを P_c に送信する。今や要求はないので、P_c は次のメッセージを待ち続ける。

コーディネータは臨界領域に入るプロセスをただ 1 つに制限するので、このアルゴリズムは相互排他を保証する。またこのアルゴリズムは、先着優先方式を使用しているので、飢餓が生じないことも保証する。このアルゴリズムは主な利点は実装が簡単である点及び、臨界領域へ入るごとに、単に 3 つのメッセージ、すなわち要求、応答、及び解放が必要である点である。しかし、集中方式の一般的な欠点を持っている。すなわち、単一のコーディネータは**単一故障点 (single point of failure)** になりうること、及び大規模システムでは性能のボトルネックとなりうる。さらには、故障処理のために、コーディネータの故障を検知するための手段や、次のコーディネータを選択するための手段や、処理を継続する前に要求待ち行列を再構築するための手段が必要である。

6.4.2 分散型の方法

分散型の方法では、相互排他のための決定は全体システムにわたって分散している。すなわち、同じ臨界領域に入ろうとするすべてのプロセスは、次にその臨界領域にどのプロセスが入るのかという結論に達する前に、互いに相談をしなければならない。そのようなアルゴリズムの最初の案が、6.3 節に述べた事象順序つけ方式に基づきに Lamport [1978] よって提案された。後に、 Ricart と Agrawala [1981] はより効率的なアルゴリズムを提案した。この方法もシス

テムにおけるすべての全順序を付けることを必要とする。相互排他のための分散アルゴリズムの一例として、**Ricart** と **Agrawala** のアルゴリズムを以下に述べる。以下の説明において、システムにおける各事象にユニークなタイムスタンプを生成するために Lamport の事象順序つけ方式が使用されていることを仮定している。

あるプロセスが臨界領域に入ろうとするとき、そのプロセスは他のすべてのプロセスに要求メッセージを送信する。メッセージは次の情報を含んでいる。

1. プロセスの識別子
2. プロセスが入ろうとしている臨界領域の名前
3. 要求メッセージのためにプロセスによって生成されたユニークなタイムスタンプ

要求メッセージを受信すると、プロセスは直ちに送信者に応答メッセージを返送するか、又は次のルールに基づき応答の送信を遅らせる。

1. もし受信プロセス自身が現在臨界領域に入っているならば、要求メッセージを待ち行列に入れて応答を遅らせる。
2. もし受信プロセスが現在臨界領域に入っていないが、臨界領域に入る順番を待っているならば、受信した要求メッセージの中のタイムスタンプと既に他のプロセスに送信した自己の要求メッセージの中のタイムスタンプと比較する。受信した要求メッセージのタイムスタンプが小さいならば、受信プロセスが要求した時よりも前に、送信プロセスが臨界領域に入ることを要求したことを意味する。それゆえ、受信プロセスは送信者に応答メッセージを直ちに返送する。他方、もし受信プロセス自身の要求メッセージが小さいタイムスタンプをもっているならば、受信者は受信した要求メッセージを待ち行列に入れて、応答メッセージの送信を遅らせる。
3. もし受信プロセスが臨界領域にもなく、臨界領域に入る順番も待っていないならば、応答メッセージを直ちに返送する。

要求メッセージを送信したプロセスは他のプロセスから応答メッセージを待ち続ける。そして、すべてのプロセスから応答メッセージを受信すると直ちに臨界領域に入る。臨界領域での実行を完了すると、待ち行列にいるすべてのプロセスに応答メッセージを送信し、それらのプロセスを待ち行列から削除する。

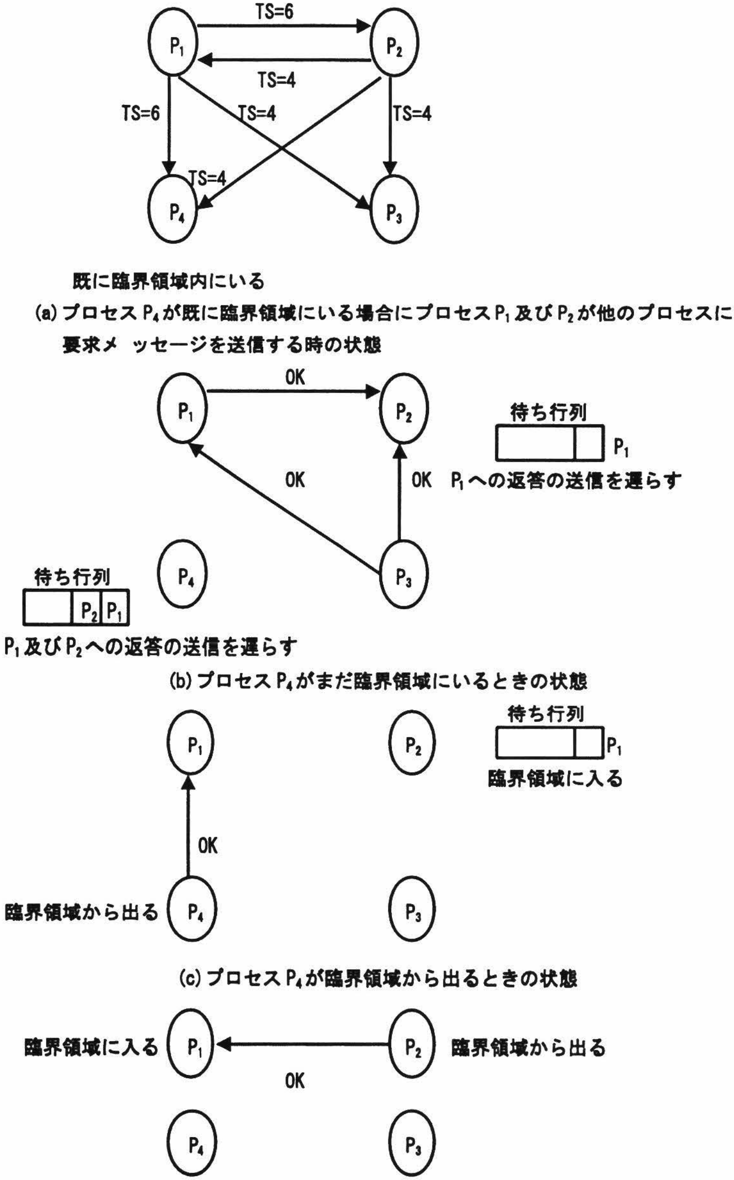
アルゴリズムがどのように動作するかを説明するために、図6.7の例を考察する。4つのプロセスP_1、P_2、P_3、P_4がある。プロセスP_4が臨界領域にある間に、プロセスP_1及びP_2が同じ臨界領域に入ろうとする。他のプロセスからの許可を得るために、プロセスP_1及びP_2がそれぞれタイムスタンプ6及び4で要求メッセージを他のプロセスに送信する(図6.7(a))。

次に図6.7(b)の状況を考えよう。プロセスP_4は既に臨界領域にあるので、P_1及びP_2への応答メッセージの送信を遅らせ、それらを待ち行列に入れる。プロセスP_3はこの時点で臨界領域には関係ないので、応答メッセージをP_1及びP_2の両者に送信する。プロセスP_2では、P_1の要求メッセージにおけるタイムスタンプ(6)よりも自己の要求メッセージにおけるタイムスタンプ(4)は小さいので、プロセスP_2はP_1への応答メッセージを送信することを遅らせ、待ち行列に入れる。他方プロセスP_1では、P_2の要求メッセージのタイムスタンプ(4)よりも自己の要求メッセージにおけるタイムスタンプ(6)が大きいので、P_1はP_2に応答を直ちに行う。

次に、図6.7(c)の状況を考える。プロセスP_4が臨界領域から出ると、P_4の待ち行列にいるすべてのプロセス(この場合、プロセスP_1及びP_2)に応答メッセージを送信する。プロセスP_2は他のすべてのプロセス(P_1、P_3及びP_4)から応答メッセージを受信したので、臨界領域に入る。しかし、プロセスP_1はプロセスP_2から応答メッセージをまだ受信していないので、待ちつづける。

最後に、プロセスP_2が臨界領域から出ると、P_2はP_1に応答メッセージを送信する(図6.7(d))。プロセスP_1はここで他のすべてのプロセスから応答メッセージを受信したので、臨界領域に入る。

このアルゴリズムは、他のすべてのプロセスから許可を得た後にのみ臨界領域に入るし、競合の場合には競合プロセスのうちただ1つのプロセスのみが他のすべてのプロセスから許可を得ることができるので、相互排他を保証する。アルゴリズムは、臨界領域に入るにはタイムスタンプの順序に従ってスケジュールされるので、飢餓も発生しない。このアルゴリズムではデッドロックも発生しないことがRicart及びAgrawala [1981]によって証明されている。さらに、n個のプロセスが存在するなら、アルゴリズムはn-1の要求メッセージ及びn-1の応答メッセージを必要とし、全体で臨界領域に入るごとに2(n-1)個のメッセージを必要とする。しかし、このアルゴリズムは、あるプロセスによって臨界領域に入ろうとする要求があると、すべてのプロセスが関与しなければならないという要件があるので次の欠点がある。



(a) プロセス P_4 が既に臨界領域にいる場合にプロセス P_1 及び P_2 が他のプロセスに要求メッセージを送信する時の状態

(b) プロセス P_4 がまだ臨界領域にいるときの状態

(c) プロセス P_4 が臨界領域から出るときの状態

(d) プロセス P_4 が臨界領域から出た後の状態

図 **6.7**　相互排他のための分散型アルゴリズムを示す例

1. システムに n 個のプロセスが存在すると、プロセスの 1 つが故障すると、全体の方式が成り立たなくなるので、このアルゴリズムは n **故障点 (n points of failure)** に依存している。これは、故障したプロセスは要求メッセージに対し応答を行わないので、そのことが許可の否定として、要求を行ったプロセスによって解釈され、その結果すべての要求プロセスは際限なく待つことになるからである。

Tanenbaum [1995] はこの問題を解決する単純な改良を提案した。この改良アルゴリズムでは、許可が直ちに得られない場合に応答メッセージの送信を遅らせる。すなわち、沈黙する代わりに、受信者は「許可否決 (permission denied)」応答メッセージを要求プロセスに送信し、後に許可が得られたとき OK メッセージを送信する。それゆえ、応答メッセージ (許可否決又は OK) はどのような場合にも直ちに要求プロセスに送信される。もし要求プロセスが固定のタイムアウト期間の間にあるプロセスから応答を受信しないならば、そのプロセスが応答するまで待ち続けるか又は、そのプロセスが故障したと結論する。要求プロセスが「許可否決」応答メッセージを 1 つ又はそれ以上のプロセスから受信すると、それらのすべてのプロセスから OK メッセージを受信するまでブロック (block) する。

2. このアルゴリズムは、各プロセスが相互排他アルゴリズムに関わるすべてのプロセスについて知っていることを必要とする。この要件は、あるグループの各プロセスはそのグループに参画したり、脱退したりする状況を常に動的に把握しなければならないので、アルゴリズムの実装を複雑にしている。すなわち、あるプロセスが、あるグループに参画するとき、そのグループにおける他のすべてのプロセスの名前を受信しなければならず、そして新しいプロセスはグループの中の他のすべてのプロセスに配信されなければならない。同様に、あるプロセスがグループから脱退又は故障の場合、そのグループのすべてのメンバーは、メンバーリストからそのプロセスを削除できるように、そのことを知らされなければならない。メンバーリストの更新は、要求及び応答メッセージが既にグループのプロセス間で交換中のとき特に困難である。それゆえ、このアルゴリズムはメンバープロセスが固定で動的に変化しないグループに対してのみ適切である。

3. このアルゴリズムにおいて、臨界領域に入ろうとするプロセスは、他のすべてのプロセスと通信を行い、すべてから許可を得た後に限り、そうすること

ができる。それゆえ、ネットワークが同時にはただ 1 つのメッセージのみを扱うことを仮定しているので、プロセスが臨界領域に入ろうとする要求を行ってから、実際に臨界領域に入るまでの待ち時間は、システムに n 個のプロセスが存在するならば、$2(n-1)$ 個のメッセージの交換に要する時間である。この待ち時間はシステムに非常に多くのプロセスが存在するならば、大きいものとなる。それゆえ、このアルゴリズムは動作中のプロセスの数が少ないシステムに適している。

このアルゴリズムに対する幾つかの改善が文献において提案されている。例えば、臨界領域に入るのに、他のすべてのプロセスの合意を得るという代わりに多数決合意 (majority consensus) の考えを使用することによって、簡単な改善が可能である [Tanenbaum 1995]。すなわち、多数決合意の考えを使用するアルゴリズムでは、他のすべてのプロセスから許可を得る代わりに、他のプロセスの過半数から許可を得ると、直ちに臨界領域に入る。このアルゴリズムでは、1 つのプロセスが、同時には単一のプロセスにのみ臨界領域に入る許可を与えることができる。このアルゴリズムに対し他に 2 つの改善が行われている [Carvalho 及び Roucairol 1983, Maekawa et al. 1987]。

6.4.3　トークンパッシング方式

このトークンパッシング方式では、相互排他はシステムにおけるプロセス間を循環する単一のトークンを使用することにより実現される。トークンはその保持者に対し、ある臨界領域に入る権利を与える特別のタイプのメッセージである。公平であるために、システムにおけるプロセスは論理的にリング構造になるように構成されており、トークンはリングに沿って同じ方向 (時計回り又は反時計回り) に 1 つのプロセスから別のプロセスへと循環する。

アルゴリズムは次のように動作する。プロセスがトークンを受信すると、そのプロセスが臨界領域に入ろうとしているかどうかを検査し、次の動作を行う。

■もし臨界領域に入ろうとしているならば、そのトークンを保持し、臨界領域に入り、そして臨界領域での処理が完了した後に臨界領域から出る。そして、トークンをリングに沿って隣のプロセスに手渡す。プロセスはトークンを受信したとき、ただ 1 つの臨界領域に入ることができるという点に注意されたい。別の臨界領域に入りたい場合には、再びトークンを獲得するまで待たねばなら

ない。

■臨界領域に入ろうとしていないならば、単にリングに沿って隣のプロセスにトークンを手渡すのみである。それゆえ、もしどのプロセスも臨界領域に入ろうとしていないならば、トークンは単にリングに沿って循環し続けるのみである。

トークンはただ1つであるので、どの時点でも、ただ1つのプロセスのみが臨界領域の中に存在できる。したがってこのアルゴリズムは相互排他を保証する。さらに、リングは単方向であり、プロセスはトークンを獲得するたびに、ただ1つの臨界領域に入ることを許可されるので、飢餓は発生しえない。このアルゴリズムでは、臨界領域に入るごとのメッセージ数は1(各プロセスが常に臨界領域に入ろうとしている)から際限のない値(どのプロセスも臨界領域に入ろうとしていない)まで変化しうる。さらには、システムに合計n個のプロセスが存在すれば、あるプロセスが臨界領域に入ろうとした時点から実際に入るまでの待ち時間は0個からn-1個のトークンパッシングメッセージを交換するのに必要な時間まで変化することになる。ゼロ個のトークンパッシングメッセージは、あるプロセスが臨界領域に入ろうとした瞬間にトークンを受信した場合であり、一方n-1個のメッセージは隣のプロセスにトークンをパスした直後に臨界領域に入ろうとする場合である。

しかし、アルゴリズムは次のタイプの障害を扱う必要がある。

1. **プロセス故障**: システムにおけるプロセス故障は論理的リングの破壊を引き起こす。そのような場合、他のプロセスの間でトークンが循環し続けることができるように、新しい論理的リングが設定されなければならない。これは、プロセスが故障したとき又は故障プロセスが故障の後に回復したとき、故障したプロセスを検出すること、及び論理的リングを動的に再構成することを必要とする。

プロセスが隣からトークンを受け取ると必ず確認メッセージを返送するということをルールにすることにより、故障プロセスの検出を容易に行うことができる。このルールによると、あるプロセスはトークンを隣に送信した後、一定のタイムアウト期間の間に、確認メッセージを受信しないとき、隣のプロセスが故障したと検出する。他方では、論理的リングの動的再構成は各プロセスで現在のリング構成を更新することにより、実現される。隣のプロセスが故障し

たことをプロセスが検出すると、故障プロセスをスキップし、その後のプロセス (実際には順番の次の動作中のプロセス) にトークンをパスすることにより、故障プロセスをグループから除外する。プロセスが回復の後に動作中となると、次の循環の時にトークンを受け取ることができるように、リングにおける自己の前の隣のプロセスに知らせる。

2. **紛失トークン**: もしトークンが紛失されると、新しいトークンが生成されねばならない。それゆえ、アルゴリズムは紛失トークンの検出及び再生機構を持たねばならない。この問題を解く 1 つの方法は、リングの上のプロセスの 1 つを「監視 (monitor)」プロセスに指定することである。監視プロセスはリング上に「トークン保持者は誰か」というメッセージを周期的に循環させる。このメッセージはリングに沿って 1 つのプロセスから別のプロセスへと循環する。このメッセージを受信したときトークンを保持しているプロセス以外は、すべてのプロセスはこのメッセージと隣のプロセスに単にパスするのみである。保持しているプロセスは隣にパスする前に、メッセージの特定のフィールドに自己の識別子を書き込む。メッセージが監視プロセスに戻ってきたとき、メッセージの中のこの特定フィールドを検査する。このフィールドに何もなければ、トークンが紛失したと判定し、新しいトークンを生成し、これをリングに循環させる。

この方法に関して 2 つの問題がある。監視プロセス自身が故障しうること、及び「トークン保持者は誰か」メッセージ自体も紛失しうるという 2 点である。両方の問題は監視プロセスを 2 つ以上にすることにより解決される。各監視プロセスはリング上のトークンの存在を独立に検査する。しかし、監視プロセスがトークンの紛失を検出すると、どちらの監視プロセスが新しいトークンを生成し循環させるかを決定するための選任 (election) を他の監視プロセスとの間で行う (選任アルゴリズムはこの章の後の部分に述べる)。選任は、各監視プロセスがトークンの紛失を独自に検出し、新しいトークンを独自に生成する場合に起こりうる複数のトークンの生成を予防するために必要である。

6.5　デッドロック

以前の節において述べたように、システムの中に幾つかの資源があり、それらの資源に対し、資源割り当て方式が、あるプロセスによる排他的なアクセスを行うことを保証しなければならない。システムは有限個の各資源タイプのユ

ニット(例えば、3台のプリンタ、6台の磁気テープ装置、4台のディスク装置、2台のCPUなど)からなっている。この状況では、ある資源を使用するための事象の順序は以下の通りである。

1. **要求 (request)**: 最初にプロセスは資源に対し要求を行う。もし、他のプロセスによって使用されているなどの理由で、要求された資源が使用できないならば、要求したプロセスは、システムによって要求資源が割り当てられるまで、待たなければならない。要求されたタイプのユニットがシステムに複数あるならば、どのユニットでも要求を満たすことに注意されたい。またプロセスは資源のユニット数を越えない範囲で必要に応じて複数の資源を要求することができることにも注意されたい。

2. **割り当て (allocate)**: システムは資源を要求プロセスに可能な限り直ちに割り当てる。システムは、各資源があいているのか割り当てられているのか、また割り当てられているならば、どのプロセスにかを記録するテーブルを維持する。もし要求された資源が別のプロセスに割り当てられているならば、要求プロセスはこの資源への待ち行列に加えられる。システムが要求プロセスに資源を割り当てると、そのプロセスは排他的にその資源を使用することができる。

3. **解放 (release)**: 割り当てられた資源をプロセスが使用し終わると、プロセスはシステムに資源を解放する。資源の現在の使用可能状態を反映させるために、割り当て及び解放の時点でシステムテーブルの記録は更新される。

資源の要求及び解放はシステム呼び出しである。システム呼び出しの例は、装置に対しては要求 (request) 及び開放 (release)、ファイルに対してはオープン (open) 及びクローズ (close)、メモリ空間に対しては割り当て (allocate) 及び開放 (free) である。3種の操作のうち、割り当て (allocate) 及び開放 (free) のみがシステムからコントロールできる操作である。他の2種の操作はプロセスによって起動される。

上記の資源の要求 (request)、割り当て (allocation)、及び解放 (release) のパターンで、あるタイプの資源に対して、もし複数の並行プロセスによってなされた合計の要求が使用できる資源の数を越えているならば、同時に資源を割り付けるための何等かの方式が必要となる。適用される方式は、デッドロック **(deadlock)** を生じないものでなければならない。デッドロックは使用可能な

数よりも多くの資源を要求するような要求がなくても、競合するプロセスが相互の進行を妨げるような状況のことである。待ち状態に入った(要求した資源が要求時点で使用可能でないので)プロセスの幾つかは、要求した資源が他の待ち状態のプロセスに保持されている場合に、再び状態を変えることがないかもしれない。この状況はデッドロック (deadlock) と呼ばれ、それに関わるプロセスはデッドロックしていると呼ばれる。デッドロックは、プロセスの集合が永久にブロックしている状態である。各プロセスは、集合の中の他のプロセスが起こすことができる事象を待っている。すべてのプロセスが待ちつづけ、そしてそれらは集合の他のどのメンバーも起こすような事象も発生させないので、集合の中のすべてのプロセスは永久にブロックしている。

デッドロックの状況の説明は例を用いて行うのが最善である。システムに2台の磁気テープ装置 T_1 及び T_2 があるとする。そして資源が空いていれば、要求者に直ちに割り当てられる割り当て方式とする。また2つの並行プロセス P_1 及び P_2 が次の順序で磁気テープ装置に要求を行うものとする。

1. P_1 が1台の磁気テープ装置を要求し、システムは T_1 を割り当てる。
2. P_2 が1台の磁気テープ装置を要求し、システムは T_2 を割り当てる。
3. P_1 がもう1台の磁気テープ装置を要求し、使用できる磁気テープ装置がないので待ち状態になる。
4. P_2 ももう1台の磁気テープ装置を要求し、使用できる磁気テープ装置がないので、これも待ち状態になる。

この時点から P_1 及び P_2 は互いに際限なく待ちつづける。P_1 は T_2 を獲得して、すなわち P_2 が T_2 を解放して、定められた処理を終わるまで T_1 を解放しない。一方 P_2 は T_1 を獲得するまで T_2 を解放しない。それゆえ、2つのプロセスはデッドロックの状態である。2つのプロセスによって行われた各要求は単に2つの磁気テープ装置を要求したのみで、システムに2つの磁気テープ装置があるので、プロセスからの要求は全体として正しい。しかし、両者のプロセスの合計要求数は磁気テープ装置の総数を越え、また資源割り当て方式は資源が空いていると要求に資源を直ちに割り当てるので、デッドロックが発生する。

デッドロックの議論において、「資源 (resource)」という用語は物理的オブジェクト(磁気テープ装置、ディスク装置、プリンタ、CPUサイクル、及びメモリ空間など)のみならず論理的オブジェクト(データベースの中のロックされたレコード、ファイル、テーブル、セマフォ、モニタなど)に対しても使用

される。しかし、これらの資源は同時には1つのプロセスのみによって排他的に使用を許可され、**先取り権無し**である場合である。先取り権無しの資源(nonpreemptable resouce) は保持しているプロセスが自発的に解放するまで、割り当てられたプロセスから資源を取り上げることができない。もし取り上げると、プロセスによって既になされた計算の悪影響が生じる。例えば、プリンタは先取り権無しの資源である。プリントを開始し、まだ完了していないプロセスからプリンタを取り上げ、別のプロセスに割り当てると、2つのプロセスからの印字出力が混合してしまう。これは明らかに受け入れ難い。

6.5.1 デッドロックの必要条件

Coffman 他. [1971] は次の条件がシステムにおいてデッドロックの生じる必要条件であると述べた。

1. **相互排他条件 (mutual-exclusion condition)**: もしあるプロセスによってある資源が保持されているならば、その資源を要求する他のプロセスは、資源が解放されるまで待たなければならない。
2. **保持待ち条件 (hold-and-wait condition)**: プロセスは現在保持している資源を解放せずに、新しい資源を要求することができる。
3. **先取り権無し条件 (no-preemption condition)**: あるプロセスに割り当てられた資源は、それを保持するプロセスによって自発的に解放された後にのみ、他のプロセスに割り当てが可能となる。
4. **循環待ち条件 (circular-wait condition)**: 2つ以上のプロセスが、各プロセスが連鎖の次のメンバーによって保持されている資源を待っているような循環連鎖を構成する。

デッドロックが生じるためには4つの条件がシステムにおいて同時に成り立たねばならない。どの1つが欠けてもデッドロックは生じない。4つの条件は、循環待ち条件は保持待ち条件でもあるので完全に独立ではない。これらの4つの条件は何等かの関連があるが、デッドロックを予防する方式を考察する際に、これらを分離して考えることは大いに有用である。

6.5.2 デッドロックモデル

デッドロックは有向グラフを用いてモデル化される。デッドロックのグラフモデルを説明する前に、グラフ理論から幾つかの用語が必要である。

1. **有向グラフ (directed graph)**: 有向グラフは (N, E) の対である。ここで N は空でないノード (node) の集合であり、E は方向を持ったエッジ (edge) である。有向エッジ (directed edge) は順序対 (ordered pair)(a, b) である、ここで a 及び b は N におけるノードである。

2. **パス (path)**: パスは 、有向エッジ (a,b), (b,c), ... ,(i,j) からなる有向グラフのノードの列 $(a,b,c,....,i,j)$ である。明らかにパスは少なくとも 2 つのノードを含んでいる。

3. **サイクル (cycle)**: サイクルは最初と最後のノードが同じのパスである。

4. **到達可能集合 (reachable set)**: ノード a の到達可能集合は a から b までのパスが存在するすべてのノード b の集合である。

5. **ノット (結び目：knot)**: ノットは空でないノードの集合 K であり、K における各ノードの到達可能集合が集合 K 自体である。ノットは常に 1 つ以上のサイクルを含む。

有向グラフの例を図 6.8 に示す。グラフはノードの集合 $\{a, b, c, d, e, f\}$ をもち、有向エッジの集合 $\{(a, b), (b, c), (c, d), (d, e), (e, f), (f, a), (e, b)\}$ をもっている。2 つのサイクル (a, b, c, d, e, f, a) 及び (b, c, d, e, b) をもっている。また、2 つのサイクルを含むノット $\{a, b, c, d, e, f\}$ をもっている。

デッドロックのモデル化では、**資源割り当てグラフ (resource allocation graph)** と呼ばれる有向グラフが使用される。そこでは、ノードの集合及びエッジの集合ともに 2 つのタイプに分けられ、次のグラフ要素となる。

1. **プロセスノード (process node)**: プロセスノードはシステムのプロセスを表す。資源割り当てグラフにおいて、通常、円形の中にプロセスの名前が書かれた円 (図 6.9 のノード P_1、P_2 及び P_3) として示される。

2. **資源ノード (resource node)**: 資源ノードはシステムの資源を表す。資源割り当てグラフにおいて、通常、長方形の中に資源の名前が書かれた長方形として示される。ある資源タイプ R_j はシステムに 1 以上のユニットがあるの

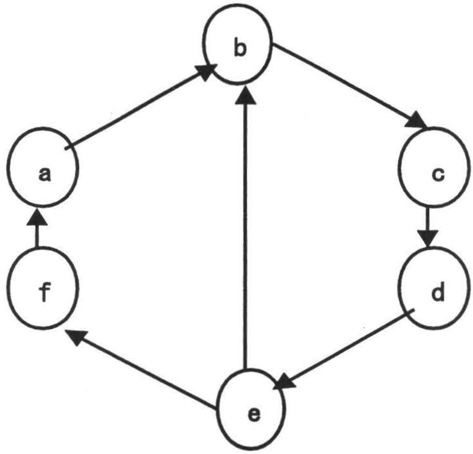


図 **6.8** 有向グラフ

で、各ユニットは長方形の中の記号 (・)(bullet) として表現される。例えば、図 6.9 の資源割り当てグラフにおいて、資源 R_1 には 2 つのユニット、R_2 には 1 つのユニット、R_3 には 3 つのユニットがある。

3. 割り付けエッジ (assignment edge): 割り付けエッジは資源ノードからプロセスノードへの有向エッジである。これは資源が現在プロセスによって保持されていることを意味する。資源タイプに複数ユニットがある場合、割り付けエッジの端は、プロセスによって 1 つのみの資源ユニットが保持されていることを示すために長方形の中の 1 つの記号 (・) に接続している。エッジ (R_1, P_1), (R_1, P_3), 及び (R_2, P_2) は図 6.9 の資源割り当てグラフにおける 3 つの割り付けエッジである。

4. 要求エッジ (request edge): 要求エッジは、プロセスノードから資源ノードへの有向エッジである。これは、資源タイプの 1 つのユニットを要求し、現在その資源を待っていることを意味する。図 6.9 の資源割り当てグラフにおいて、エッジ (P_1, R_2) 及び (P_2, R_1) は 2 つの要求エッジである。

資源割り当てグラフの構築

資源割り当てグラフはシステムにおける多くの資源について保持しているプ

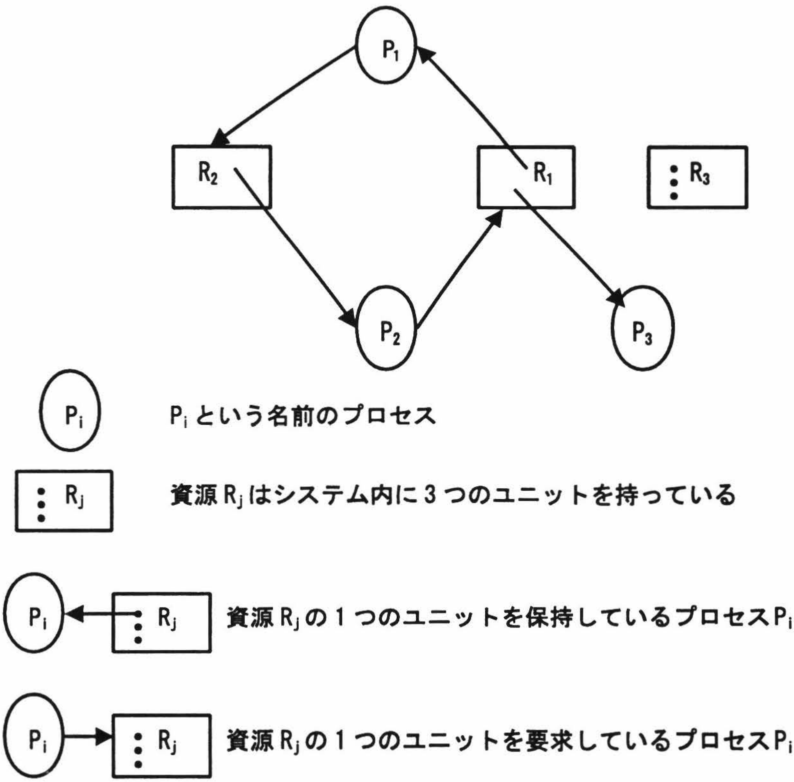

P1
R2
R1
R3
P2
P3
Pi
Pi という名前のプロセス
Rj
資源 Rj はシステム内に 3 つのユニットを持っている
Pi
Rj
資源 Rj の 1 つのユニットを保持しているプロセス Pi
Pi
Rj
資源 Rj の 1 つのユニットを要求しているプロセス Pi

図 6.9　資源配置グラフ

ロセス、及び待っているプロセスの全体像を提供する。それゆえ、システムのプロセスが資源を要求したり、解放したり、又はシステムがプロセスに資源を割り当てるに従ってグラフは動的に変化する。すなわち、プロセス P_i が資源タイプ R_j の1つのユニットを要求すると、要求エッジ (P_i, R_j) が資源割り当てグラフに挿入される。この要求が満たされると、資源 R_j の1つのユニットが P_i に割り当てられ、要求エッジ (P_i, R_j) は直ちに割り付けエッジ (R_j, P_i) に変更される。後に、P_i が R_j を解放すると、割り付けエッジ (R_j, P_i) はグラフから削除される。

多くのシステムにおいて、資源割り当てグラフは上記のように作成されるわけではない。むしろ、資源割り当て決定がデッドロックにならないようにするための道具として使用される。これらのシステムでは、使用可能な資源に対して、異なる要求／解放の順序がステップバイステップでシミュレートされ、各ステップごとにグラフはデッドロックに対して検査される。この後に、資源割り当て方式において、デッドロックを起こさない順序のみが許可される。それゆえ、これらのシステムにおいて、資源割り当てグラフは基本的に、デッドロック無しの資源割り当て方式を定式化するために、使用される。

デッドロックの必要十分条件

資源割り当てグラフにおいて、サイクルはデッドロックが存在するための必要条件である。すなわち、もしグラフがサイクルをもっていないならば、デッドロックは生じない状態を表す。他方、もしグラフがサイクルを含んでいるならば、デッドロックは存在するかもしれない。それゆえ、一般的な資源割り当てグラフにおいてサイクルの存在はデッドロックの存在のための必要条件であるが十分条件ではない。例えば、図 6.9 の資源割り当てグラフはサイクル $(P_1, R_2, P_2, R_1, P_1)$ を含んでいるが、デッドロック状態を表してはいない。これは P_3 が R_1 の使用を終わって解放すると、R_1 は P_2 に割り当てられるからである。R_1 及び R_2 の両方が P_2 に割り当てられると、P_2 は処理を完了することができ、その後 R_1 も R_2 も解放する。R_2 が解放されるとすぐに、R_1 は P_1 に割り当てられる。それゆえ、すべてのプロセスは一つずつ処理を完了することができる。

デッドロックの十分条件は以下の場合ごとによって異なる。

1. もしサイクルを構成し、プロセスによって要求されるすべての資源タイプがそれぞれ1つのユニットの場合には、グラフのサイクルはデッドロックの必

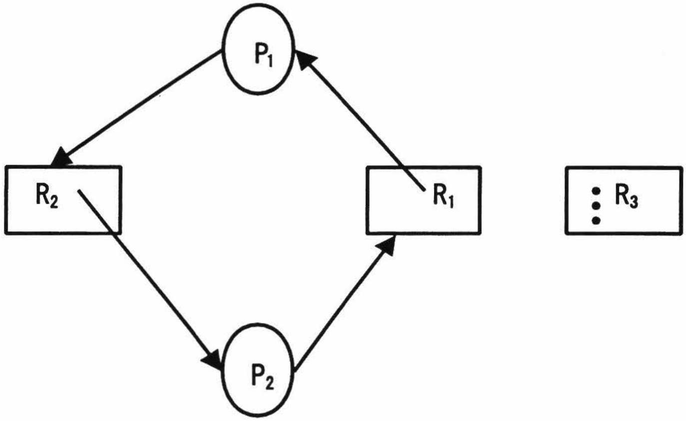


図 6.10　デッドロックを表すサイクル

要十分条件である。

例えば、図 6.10 の資源割り当てグラフはプロセス P_1 及び P_2 がデッドロックである状態を示す。このグラフにおいて、R_3 には 3 つのユニットがあるが、R_3 はサイクル (P_1, R_2, P_2, R_1, P_1) には含まれていないサイクルに含まれる R_1 及び R_2 はそれぞれ 1 つのユニットである。それゆえ、このサイクルはデッドロック状態を表す。

2. もしサイクルを構成するプロセスによって要求された 1 つ以上の資源タイプが 1 つより多くのユニットをもっているならば、グラフのサイクルはデッドロックの必要条件であるが、十分条件ではない。この場合、ノットがデッドロックの十分条件となる。

図 6.9 のグラフにおけるサイクル (P_1, R_2, P_2, R_1, P_1) はデッドロックを表していないことを既に述べた。これは資源タイプ R_1 が 2 つのユニットをもち、グラフにノットがないからである。同じグラフで P_3 が R_2 を要求し、要求エッジ (P_3,R_2) がグラフに追加された場合を考える。修正されたグラフは図 6.11 に示される。このグラフは 2 つのサイクル (P_1, R_2, P_2, R_1, P_1) 及び (P_3, R_2, P_2, R_1, P_3) を持っている、また 1 つのノット $\{P_1, P_2, P_3, R_1, R_2\}$ を持っている。グラフはノットを含んでいるので、P_1、P_2 及び P_3 がデッドロックしている状態を表す。

資源割り当てグラフに関して、デッドロックの必要十分条件は以下のように

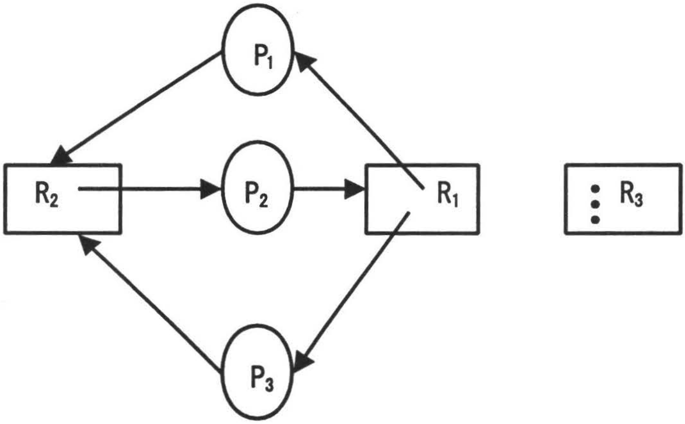


図 6.11 デッドロックを表すノット

要約できる。

■サイクルはデッドロックの必要条件である。

■もしサイクルに含まれる各資源タイプにユニットが1つのみである場合には、サイクルはデッドロックが存在するための必要十分条件である。

■もしサイクルに含まれる1つ以上の資源タイプが1より多いユニットをもっているならば、ノットがデッドロックが存在するための必要十分条件である。

待ち状態グラフ

すべての資源タイプがそれぞれ1つのユニットのみの場合には、資源割り当てグラフの簡略形が通常使用される。簡略化グラフは、元の資源割り当てグラフから資源ノードを取り除き、関連するエッジを簡略化したものである。この簡略化は、資源は常に現在の所有者 (資源を保持しているプロセス) によって特定できるということに基づいている。図 6.12 は資源割り当てグラフとその簡略形を示す。

簡略化グラフは通常、どのプロセスが他のどのプロセスを待っているかを明示しているので、**待ち状態グラフ (wait-for graph：WFG)** として知られている。例えば、図 6.12(b) の WFG において、プロセス P_1 及び P_3 はプロセス P_2 を待っており、プロセス P_2 は P_1 を待っている。WFG は各資源タイプに1つのみのユニットをもつ場合に、すなわち、サイクルが WFG においてデッド

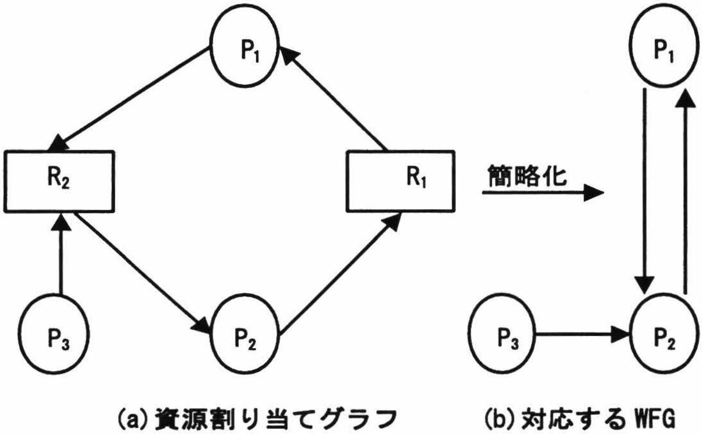


図 6.12　資源配置グラフから WFG への変換

ロックのための必要十分条件である場合に作成される。

6.5.3　分散システムにおけるデッドロックの扱い

基本的に分散システムにおけるデッドロックは集中システムにおけるデッドロックに類似している。それゆえ、デッドロックに関する上記の記述は集中システム又は分散システムの両方に当てはまる。しかし、資源、プロセス、及びその他の情報がシステムの中の異なるノードに分散されているので、分散システムにおけるデッドロックの扱いの方が集中システムにおけるよりも複雑である。

デッドロックを扱うために通常 3 つの方策が使用される。

1. **回避 (avoidance)**: 資源はデッドロックを回避するために注意深く割り当てられる。

2. **予防 (prevention)**: デッドロックを予防するために、プロセスが資源を要求する方法に制約を設ける。

3. **検出と回復 (detection and recovery)**: デッドロックの発生を許容し、それらを検出するための検出アルゴリズムが使用される。デッドロックが検出された後に、ある種の方法で解消される。

3 番目の方法が分散システムでもっとも一般的に用いられるが、他の 2 つの

方法も簡単に述べる。

分散したデッドロックを2種類に区別する人々もいることに注意されたい。それらは**資源デッドロック (resource deadlock)** 及び**通信デッドロック (communication deadlock)** である。既に述べたように、資源デッドロックは2つ以上のプロセスが互いに保持している資源を永久に待つことにより発生する。他方、通信デッドロックはプロセスの集合の間で発生する。通信デッドロックでは、処理を開始しようとしているがプロセス間を通過するメッセージがなくて、集合の他のプロセスからのメッセージを待ってプロセスがブロックしている状態である。集合のどのプロセス対の間でも通過するメッセージがない場合、どのプロセスもメッセージを受信しないであろう。これは集合のすべてのプロセスがデッドロックしていることを意味する。どのプロセスが他のどのプロセスからのメッセージの受信を待っているかを示すために、通信デッドロックは WFG を用いて容易にモデル化される。したがって、通信デッドロックの検出は、各資源タイプにただ1つのユニットがある場合の検出と同様の方法で行われる。

デッドロックの回避

デッドロック回避方法は、デッドロックになるような割り当てを避けるためにシステムの将来状態を予測するといった、プロセスの資源使用に関する事前知識を用いる。デッドロック回避アルゴリズムは通常、次のステップからなる。

1. プロセスが資源を要求するとき、資源が割り当てられる状態にあっても、プロセスに資源を直ちに割り当てることはしない。システムは単に要求が認められたと仮定するのみである。

2. ステップ1でなされた仮定及びプロセスの資源使用に関する知識を用いて、プロセスの要求を認めることは安全 (safe) か不安全 (unsafe) かの解析を行う。

3. ステップ2での解析結果が、資源の割り当てが安全である場合にのみ、プロセスに資源を割り当てる。それ以外には要求は遅らされる。

デッドロック回避のアルゴリズムは安全及び不安全の概念に基づいているので、資源割り当てにおいて安全性について理解することが重要である。システテ

ムがデッドロック状態になく、またプロセスからの資源要求がすべて完結できるようなプロセスの順序が存在するならば、システムは**安全状態 (safe state)** にあると言われる。特定の安全状態に対して、多くのプロセス順序が存在するであろう。すべてのプロセスの完結を保証するようなプロセス順序は、**安全シーケンス (safe sequence)** と呼ばれる。安全シーケンスの定式化は、次の条件を満たす考えに基づいている。安全シーケンスにあるどのプロセス P_i に対しても、P_i が要求できる資源は、現在使用可能な資源であって、安全シーケンスにおけるプロセスで、P_i 以前のプロセスによって保持されている資源であるという条件を満たす資源である。この条件はプロセス P_i が完結することを保証する。もし P_i が必要とする資源が直ちに使用できないならば、P_i はシーケンスの P_i より前にいる他のプロセスが完結するまで待つことができる。それらが完結したとき、P_i は必要な資源をすべて獲得し、処理を完結することができる。もし安全シーケンスが存在しないならば、システム状態は**不安全 (unsafe) 状態**と言われる。

安全状態及び不安全状態の概念を例を用いて説明する。システムにおいて、特定の資源タイプのユニットが総数 8 つあり、P_1、P_2 及び P_3 の 3 つのプロセスが競合すると仮定する。P_1、P_2 及び P_3 によって要求される資源の最大ユニット数は、それぞれ 4、5、及び 6 とする。また、それぞれの 3 つのプロセスは資源を 2 ユニット保持しているとする。それゆえ、システムの現在状態において、資源の 2 つのユニットが空いている。システムの現在状態は図 6.13(a) に示されるようにモデル化される。

図 6.13(a) の状態が安全か否かを判定してみる。図 6.13 において行われる解析は、すべてのプロセスが完結できるシーケンスが存在するので、この状態が安全であることを示している。実際、図に示したように、この状態に対して、2 つの安全シーケンス (P_1, P_2, P_3) 及び (P_1, P_3, P_2) が存在する。これらの 2 つの安全シーケンスの前者に対し、資源ユニットのスケジュールを考察する。図 6.13(a) の状態からスタートして、現在空いている 2 つのユニットを P_1 が要求し、獲得するまで、スケジューラは P_1 を排他的に走らせることができる。その結果図 6.13(b) になる。P_1 が完結し、保持していた資源を解放すると、図 6.13(c) の状態になる。それから、スケジューラは P_2 を選び走らせる。その結果、図 6.13(d) の状態になる。P_2 が完結し、保持していた資源を解放すると、システムは図 6.13(e) の状態になる。次に、使用可能な資源を使用して、P_3 が走り完結する。図 6.13(a) の最初の状態は、システムが注意してスケジュールすれば、デッドロックを回避できるので、安全である。この例は、特定の状態

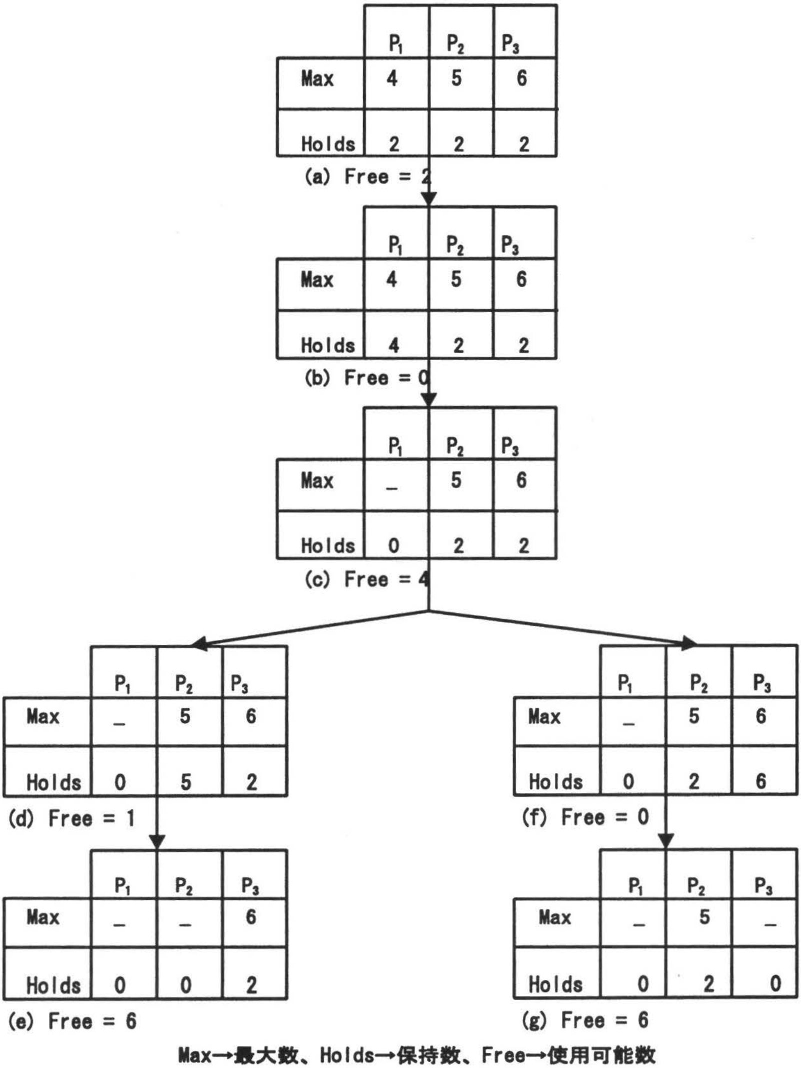


図 6.13 (a) の状態が安全であり、2 つの安全シーケンスを持っていることを示す。

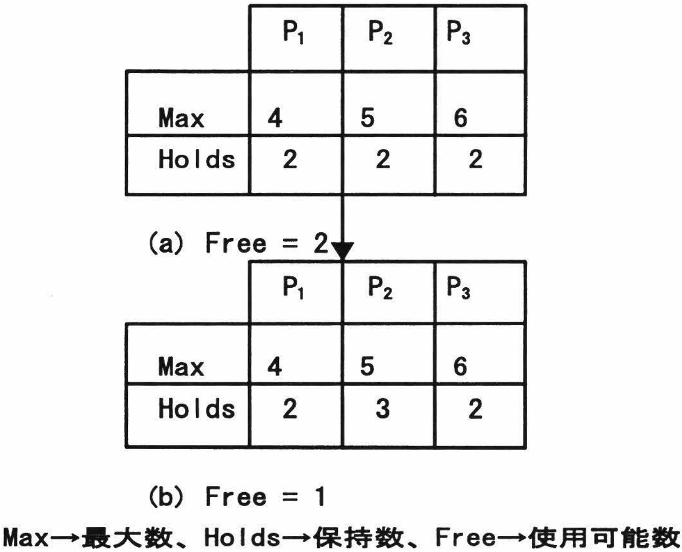


図 **6.14**　割り当てによってはシステムが安全から不安全になることを示す。

に対して 2 つ以上の安全シーケンスがありうることを示している。

もし資源割り当てが注意深くなされないと、システムは安全状態から不安全状態に移行するかもしれない。例えば、図 6.14 の例を考える。図 6.14(a) は図 6.13(a) の初期状態と同じ初期状態である。この場合に、P_2 が資源のユニットをもう 1 つ要求し、システムによってそれが割り当てられたとする。その結果システムの状態は図 6.14(b) に示す状態となる。このとき空いている資源は 1 ユニットのみであり、3 つのプロセスが完結するのには不十分であるので、システムはもはや安全状態にはない。それゆえ、図 6.14(b) の状態に対して、安全シーケンスは存在しない。したがって、P_2 へ資源の 1 ユニットを割り当てるという決定はシステムを安全状態から不安全状態に移行させる。

安全状態に関して次の点に注意することが重要である。

1. どの資源もまだ割り当てられておらず、すべてが使用可能 (空いている) な初期状態は常に安全状態である。

2. 安全状態から、すべてのプロセスが完結することをシステムは保証できる。

3. 不安全状態はデッドロック状態ではないが、デッドロック状態になりがちである。すなわち、不安全状態から、すべてのプロセスが完結することをシス

テムは保証できない。

デッドロック回避アルゴリズムは基本的に、システムが常に安全状態に留まるように、資源割り当てを行う。システムの初期状態は常に安全状態であるので、あるプロセスが現在使用可能な資源を要求するとき常に、システムはプロセスへの資源割り当てがシステムの状態を安全状態から不安全状態に変更しないかを検出するための検査を行う。もし不安全状態にならないならば、要求は直ちに認められる。そうでなければ要求は遅らされる。

デッドロック回避アルゴリズムは理論的には魅力があるが、次に述べる理由により実際には余り使用されない。

1. これらのアルゴリズムは、多くのプロセスの資源要求に関する事前知識を前提として動作する。しかし、実際には、プロセスが最大どれだけの資源を要求するかは事前にはほとんど不明である。最近のオペレーティングシステムは、一層ユーザフレンドリーなインタフェースを提供しようとしている。その結果、ユーザは必要とする資源について、考えなくなっている。

2. これらのアルゴリズムはまた、特定の資源を使用し完結するプロセスの数が固定で既知であることも仮定している。しかし、実際には、プロセスの数は固定ではなく新しいユーザがログインしたりログアウトしたりして動的に変化する。

3. これらのアルゴリズムはまた、特定の資源のユニット数が常に固定で既知であることも仮定している。しかし、実際には、1ユニット以上の突然の故障や修理によって、ユニットの実際の数は動的に変化する。

4. これらのアルゴリズムが動作する仕方は資源割り当てを余りにも厳密に制限するので、システム性能を相当減少させる。これは、アルゴリズムは起こりうる最悪のケースを想定し、その場合にもデッドロックにならないことを保証するからである。この最悪状況は起き得るが希有である。このようにして、多くの安全な要求も待たされる。

デッドロック回避アルゴリズムの実際の制約は、分散システムにおいては資源割り当て決定のための情報の収集は困難で非効率的であるので、さらに厳し

くなる。それゆえ、デッドロック回避方策は分散オペレーティングシステムにおいては決して使用されない。

デッドロック予防

この方法はデッドロックが不可能であるようにシステムを設計するという考えに基づく。この方法は、回避及び検出とは、割り当てに関して実行時に行う必要がないという点で異なる。

相互排他、保持待ち、先取り権無し、及び循環待ちはシステムにデッドロックが発生するための 4 つの必要条件であることは既に述べた。それゆえ、これらの条件の少なくとも 1 つが決して満たされないことを何等かの方法で確証できれば、デッドロックは発生しないであろう。この考えに基づき、3 つの重要なデッドロック予防方法がある。一括要求 (collective request)、順序要求 (ordered request) 及び先取り権付き要求 (preemption) である。一括要求は保持待ち条件を否定する。順序要求は循環待ち条件を否定する。先取り権付きは先取り権無し条件を否定する。

相互排他条件もまた、ある種の非共有資源に対して、それらを使用する代替の方法を工夫することにより、否定できる。例えば、 [Tanenbaum 1992] からの次の例はプリンタについてこの考えの適用を示している。プリンタは非共有資源である。しかし、プリンタ出力をスプールすることにより、複数のプロセスが同時に出力を生成することができる。スプールされた出力はプリンタデーモンによって一つずつプリンタに転送される。それゆえ、プリンタデーモンは、物理的にプリンタに要求を行うただ 1 つのプロセスである。デーモンは他の資源に要求を決して行わないので、プリンタに関するデッドロックは構造的に不可能となる。不幸にも、一般に、ある種の資源は本質的に非共有であり、それらを使用するための代替方法を工夫することができないので、相互排他条件を否定することによりデッドロックを予防するという方法は不可能である。それゆえ、デッドロック予防のための相互排他条件の否定はほとんど使用されない。他の 3 つの方法はより一般的に使用される。以下にそれらについて述べる。

一括要求

この方法は、プロセスが資源を要求するとき常に、他の資源を保持しないことを保証することにより、保持待ち条件を否定する。次の資源割り当て方策の一つがこれを保証するために使用される。

1. プロセスは処理をはじめる前に必要な資源すべてを要求する。もしすべての必要な資源が使用可能ならば、それらはすべてプロセスに割り当てられ、プロセスは完結するまで動作する。もし要求された資源が1つ以上使用不可能ならば、資源は1つも割り当てられずに、プロセスはただ待つのみである。

2. 処理が始まる前にすべての資源を要求するのではなく、他の資源を保持していないときに限り資源を要求するというルールに従うならば、プロセスは実行中に資源を要求することができる。もしプロセスが既に資源を保持しているならば、上記のルールに対して、すべての資源を解放し、それから再びすべての必要な資源を要求するというルールに変更することもできる。

第二の方策は最初のに比べ、次の利点をもっている。

1. 実際に、ほとんどのプロセスは走りはじめるまでにどれだけ多くの資源を必要とするかを意識しない。そのような場合、第二の方策が有用である。

2. 長いプロセスは実行の終わりのころに、幾つかの資源を要求するかもしれない。最初の方策では、全体の実行の期間、これらの資源を不必要に保持することになる。第二の方策では、しかし、プロセスはこれらの資源を必要なときのみ要求することになる。

デッドロック予防の一括要求方法は簡単で効率的であるが、次の問題がある。

1. 低い資源稼働率、すなわち必要以上に占有されているという問題がある。その理由は、プロセスは多くの資源を保持するが、それらの幾つかはかなりの長期間にわたって実際に使用されないからである。

2. プロセスは多くの資源を必要とするが、必要な資源を要求するとき、1つ以上の資源が使用不可能であることもあるので、プロセスの飢餓をまねくかもしれない。

3. この方法はまた、課金の問題がある。プロセスが必要でない長期間の間資源を保持するとき、アイドル状態の資源に対してだれが料金を負担するかという問題がある。

順序要求

この方法では、すべての資源タイプの全体順序を定めるためのユニーク全体番号 (unique global number) が各資源タイプに割り付けられている。資源割り当て方策は、プロセスはいつでも資源を要求できるが、プロセスは既に保持しているどの資源の番号よりも小さい番号の資源を要求してはならないということに従って使用される。すなわち、プロセスが番号が i の資源タイプを保持しているならば、$j > i$ である番号 j の資源タイプのみを要求できる。もしプロセスが同じ資源タイプの複数のユニットを必要とするならば、すべてのユニットに対して 1 つの要求として発行しなければならない。このルールによると、資源割り当てグラフはサイクルを持ち得ない (すなわち循環待ち条件を否定する) ことが証明されている。したがってデッドロックは起きない。

このアルゴリズムは、プロセスはすべての資源を厳密に昇べきの順で獲得することを要求しないことに注意されたい。例えば、3 番及び 7 番の 2 つの資源を保持しているプロセスが 5 番の資源を要求する前に 7 番の資源を解放するかもしれない。プロセスが 5 番の資源を要求するとき、5 番より大きい番号の資源を保持していないので、この要求は認められる。

資源の順番は資源の自然の使用パターンにしたがって決定される。例えば、通常、磁気テープ装置はプリンタの前に必要とされるので、磁気テープ装置にプリンタよりも小さい番号を附することは合理的であろう。しかし、自然の順番は必ずしもすべての処理について同じではない。それゆえ、決定された順番にあっている処理は資源を効率的に使用することが期待できるが、他の処理では資源を浪費することにもなりかねない。別の問題は、一度順番が決定されると、順番はプログラムに組み込まれるので長期間にわたって同じである。順番の変更は複数のジョブについて、再プログラミングを必要とする。しかし、順番の変更はシステムに新しい資源が加わったときに不可避となろう。これらの問題にも拘わらず、順序要求の方法がデッドロックを扱う最も効率的な方法の 1 つである。

先取り権

先取り権ありの (preemptable) 資源は、その状態を容易にセーブ (保存) したりリストア (復元) したりできる資源である。そのような資源は、今までプロセスによって行われてきた計算に何等の悪影響を与えることなく、現在割り当てられているプロセスから一時的に資源を取り去ることができる。CPU のレジスタや主メモリはこのような先取り権ありの資源の例である。もし資源が先取

り権ありならば、デッドロックは、先取り権なし条件を否定する次の資源割り当て方策のいずれかを使用することにより予防することができる。

1. プロセスが、現在使用不可能の資源を要求すると、プロセスによって保持されているすべての資源がプロセスから取り去られ (先取りされ)、プロセスはブロックする。そのプロセスによって要求されていた資源及び先取りされていた資源が使用可能となり、そのプロセスに割り当てられると、そのプロセスのブロックは解除される。

2. プロセスが現在使用不可能な資源を要求すると、システムは、その資源が他の資源を待つために現在ブロック中であるプロセスによって保持されているか検査する。もしそうであるならば、要求された資源は待っているプロセスから取り上げられ (先取りされ)、要求したプロセスに与えられる。そうでなければ、要求したプロセスはブロックされ、要求資源が使用可能になるまで待つ。このプロセスが要求した資源の割り当てを待ってブロックしている間に、このプロセスが既に保持している資源の幾つかは、取り上げられる (先取りされる) かもしれない。要求された資源及び先取りされた資源が使用可能になり、プロセスに割り当てできるようになると、プロセスのブロックは解除される。

一般に、デッドロック予防のためのこの方法の適用性は、先取り権ありの資源に対してのみ有効であるので、極端に制限される。しかし、原子トランザクション及び全体タイムスタンプの使用により、この方法は分散データベーストランザクション処理システムにおけるデッドロック予防に有効となる。トランザクション機構では、どのような悪影響もなく、トランザクション (プロセス) を強制終了する (死なせる) ことができる (トランザクションは第 9 章に述べる)。これにより、弊害なく、資源を保持しているプロセスから資源を先取りすることができる。

トランザクションに基づくデッドロック予防方法において、各トランザクションはユニーク優先番号 (unique priority number) をシステムによって割り付けられている。そして 2 つ以上のトランザクションが同じ資源を競合すると、競合解除 (break the tie) のために、その優先番号が使用される。例えば、Lamport のアルゴリズムが、システム側の全体ユニークタイムスタンプを生成するために使用され、各トランザクションは生成されるときにユニークタイムスタンプを割り付けられるであろう。トランザクションのタイムスタンプは

優先番号として働く。すなわち、小さいタイムスタンプの値をもつトランザクションは、その方がより古いので、高い優先度をもつであろう。

Rosenkrants その他 [1978] は、この考えに基づく次のデッドロック予防方式を提案した。

1. **待機消滅方式 (wait-die scheme)**:この方式において、トランザクション T_i が現在別のトランザクション T_j によって保持されている資源を要求すると、もし T_i のタイムスタンプが T_j のタイムスタンプよりも小さいならば、T_i はブロックする (待つ)。それ以外ならば、強制終了させられる (死ぬ)。例えば、3 つのトランザクション T_1、T_2、及び T_3 があり、T_1 が最も古く (最も小さいタイムスタンプ値をもつ)、T_3 が最も新しい (最も大きいタイムスタンプ値をもつ) とする。もし現在 T_2 が保持している資源を T_1 が要求すると、T_2 によって資源が自発的に解放されるまで、T_1 はブロックし待つ。他方、もし T_3 が T_2 によって保持されている資源を要求すると、T_3 は強制終了させられる。

2. **待機負傷方式 (wait-wound scheme)**:この方式では、もしトランザクション T_i が現在別のトランザクション T_j によって保持されている資源を要求し、そのタイムスタンプが T_j のタイムスタンプよりも大きいならば、T_i はブロックする (待つ)。そうでなければ、T_j は強制終了させられる (T_i によって負傷させられる)。上記と同じトランザクション T_1、T_2、及び T_3 の例を考える。もし T_1 が T_2 によって保持されている資源を要求すると、T_2 を強制終了させることにより資源は先取りされ、T_1 に与えられる。他方、もし T_3 が T_2 によって保持されている資源を要求すると、資源が T_2 によって自発的に解放されるまで、T_3 はブロックし、待つことになる。

両方の方式ともに古いトランザクションを優先している。その理由は、一般に、新しいトランザクションよりも古いトランザクションほど長期間動作し、システム資源を多く使用してきているからである。しかし、両方式の新しいトランザクションの取り扱いの仕方に注意することは価値がある。待機消滅方式では、新しいトランザクションは古いトランザクションによって保持されている資源を要求するとき、強制終了させられる。強制終了させられたトランザクションは定められた時間の後に再開始させられ、古いトランザクションが資源をまだ保持していると、再び強制終了させられる。このサイクルは新しいトランザクションが実際に資源を獲得するまで繰り返される。待機消滅方式のこの

問題は、要求された資源が使用可能になったときのみ、強制終了されたトランザクションを再開始するという実装機構を使用することによって解決される。

他方、待機負傷方式において、新しいトランザクションが古いトランザクションによって強制終了させられる(負傷させられる)場合に、定められた時間の後に再開始させられ、この時点でもし古いトランザクションによって先取りされた資源がまだ保持されているならば、このトランザクションはブロックさせられる。それゆえ、待機負傷方式の実装は、待機消滅方式よりも簡単である。さらには、飢餓を避けるために、両方式の実装において、強制終了の後に再開始されるトランザクションは新しいタイムスタンプを割り付けてはならない(新しいトランザクションは時間の経過とともに古いトランザクションとなり、強制終了を繰り返されることがない)という点は重要である。

デッドロック検出

デッドロックを扱うこの方法では、システムはデッドロックを予防するどのような試みも行わず、プロセスが資源を要求し、制限なしに互いに待つままにする。その代わり、デッドロックが発生したか否かを判断するためにシステムの状態を調べ続ける。デッドロックが検出されると、システムはデッドロックから回復するためのアクションを行う。分散システムにおいてデッドロック検出のための方法を以下に述べる。またデッドロック状態から回復するための方法を次の節で述べる。

基本的に、デッドロック検出アルゴリズムは集中システム及び分散システムの両者において同様である。アルゴリズムは、資源割り当てグラフの形で多くのプロセスへの資源の割り当てに関する情報の維持及び資源システムが各資源タイプに単一/複数のユニットをもつかどうかに依存して、グラフにサイクル/ノットがあるかの探索に基づいている。しかし、簡単のために、次の説明では、各資源タイプにただ1つのユニットの場合を考察する。それゆえ、デッドロック検出アルゴリズムは、WFGを維持し、WFGにサイクルの有無を検出することに簡略化される。

次のステップは分散システムにおいてWFGを構築するためのステップである。

1. 次の方法で、システムの各サイトに対して個別のWFGを作成する。図6.9の方法により、このサイトに存在するすべての資源に対し資源割り当てグ

ラフを作成する。すなわち、このサイトの資源割り当てグラフにおいて、資源ノードはすべてのローカル資源に対して存在し、プロセスノードは、このプロセスがローカルであるか非ローカルであるかに関わらず、このサイトの資源を保持又は待っているすべてのプロセスに対して存在する。

2. ステップ 1 で作成された資源割り当てグラフを、資源ノードを除き、エッジを集約して対応する WFG に変換する。ステップ 1 及びこのステップ 2 は説明を明確にするためにここでは区別されている。実際のアルゴリズムは直接 WFG を作成するように設計されるであろう。

3. すべてのサイトの WFG を合成し、1 つの全体 WFG を作成する。

図 6.15 に示された単純な例を用いて、手続きを説明する。システムは 2 つのサイト (S_1 及び S_2) のみからなり、S_1 は 2 つの資源 R_1 及び R_2 をもち、S_2 は 1 つの資源 R_3 をもつものとする。また、3 つのプロセス (P_1、P_2、P_3) があり、次のように 3 つの資源を競合しているものとする。

- P_1 は R_1 を保持し、R_3 を要求する
- P_2 は R_2 を保持し、R_1 を要求する
- P_3 は R_3 を保持し、R_2 を要求する

2 つのサイトに対する資源割り当てグラフは図 6.15(a) に示される。プロセス P_1 及び P_3 は、両方のサイト上の資源を要求したので、両方のサイトに現れる。他方、プロセス P_2 によって要求された両方の資源は S_1 上にあるので、プロセス P_2 はサイト S_1 上のグラフにのみ現れる。

図 6.15(b) は 2 つのサイトにおける対応する WFG を示し、図 6.15(c) は 2 つのサイトのローカル WFG を合併することにより得られる全体 WFG を示す。2 つのサイトのローカル WFG にはサイクルを含んでいないが、全体 WFG にはサイクルを含み、システムがデッドロック状態にあることを意味していることに注意されたい。それゆえ、ローカル WFG は分散システムにおいてすべてのデッドロックを表すには不十分であり、すべてのローカル WFG の合併 (union) をとることによって得られる全体 WFG の作成が、システムがデッドロック状態にあるのか否かの最終決定を行うのに必要であることをこの例は示している。

分散システムにおいてデッドロック検出を実装するのに主な困難は、どう

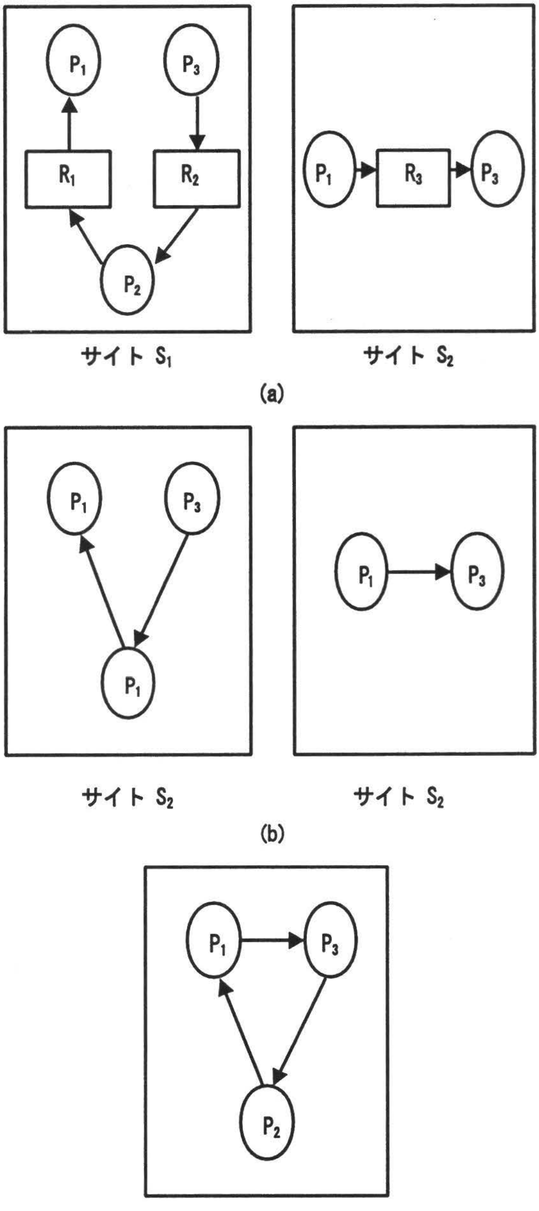


図 6.15 分散システムにおける WFG の構築。(a) 各サイトの資源割り当てグラフ、(b) (a) のグラフに対応する WFG、(c) (b) の 2 つのローカル WFG を合併した全体WＦＧ

やって WFG を維持するかである。分散システムにおいて WFG を組織化するための 3 つの一般的技法は、集中型 (centralized)、階層型 (hierarchical)、及び分散型 (distributed) の技法である。これらの技法を次に述べる。しかし、それを述べる前に、デッドロック検出アルゴリズムの最も重要な特性の 1 つは正当性 (correctness) であることに注意されたい。正当性は次の特性に依存している [Knapp 1987]。

1. **プログレス特性 (progress property)**:この特性は、すべてのデッドロックは有限時間以内に検出されなければならないことを意味する。

2. **安全特性 (safety property)**:もしデッドロックが検出されると、実際に存在していなければならない。メッセージ遅れや更新遅れの WFG は時には偽サイクル (false cycle) として検出され、その結果、実際に存在しないデッドロックの検出となる。そのようなデッドロックは**幽霊デッドロック (phantom deadlock)** と呼ばれる。

デッドロック検出の集中型方式。 集中型のデッドロック検出方式では、各サイトにはローカル資源のための WFG を維持しているローカルのコーディネータが存在し、すべての個別の WFG の合併を行う**中央コーディネータ (central coordinator)**(集中型デッドロック検出者としても知られている) が存在する。中央コーディネータはすべてのサイトのローカルコーディネータから情報を受け取り全体 WFG を構築する。この方式では、デッドロック検出は次のようにして行われる。

1. もしローカル WFG にサイクルが存在するならば、ローカルデッドロックが存在することを表す。そのようなデッドロックはサイトのローカルコーディネータによってローカルに検出され、解決される。

2. 2 つ以上のサイトの資源を含むデッドロックは全体 WFG におけるサイクルとして検出される。それゆえ、このようなデッドロックは中央コーディネータによって検出され、解決される。

集中型の方式において、ローカルコーディネータはローカル状態情報をメッセージの形式で中央コーディネータに送信する。次に述べる方法の 1 つがロー

カルコーディネータから中央コーディネータへの情報転送に使用される。

1. **連続的転送 (continuous transfer)**: ローカルコーディネータは、ローカル WFG に新しいエッジが追加されたり、削除されたりするとき更新情報のメッセージを送信する。

2. **周期的転送 (periodic transfer)**: メッセージの数を減少させるために、ローカルコーディネータは、前回送信時以来に、ローカル WFG に追加、又は削除されたエッジのリストを周期的に (ローカル WFG における変化が一定数発生したとき) 送信する。

3. **要求時転送 (transfer-on-request)**: 中央コーディネータが要求するときのみ、ローカルコーディネータは前回送信して以来の、ローカル WFG に追加又は削除されたエッジのリストを送信する。この場合、中央コーディネータはサイクル検出アルゴリズムを周期的に起動する。この起動の直前に各サイトからの情報を要求する。

集中型デッドロック検出方式は概念的には単純であるが、幾つかの欠点がある。第1に、中央コーディネータの故障に対して影響を受けやすい。したがって、このような故障に対する特別の対策が必要である。1つの方法は、中央コーディネータの処理を重複させるバックアップの中央コーディネータを準備することである。第2に、非常に多くのサイトをもつ大規模システムでは、中央コーディネータは性能の隘路となることもある。第3に、中央コーディネータは偽デッドロックを検出するかもしれない。以下に単純な例を用いて、このアルゴリズムがどのように偽デッドロックを検出するかを説明し、それからこの第3の欠点に対する対策方法を述べる。

図 6.15 の構成と同じ構成を考える。今回は次にのべる順番で、3つの資源 (R_1, R_2, R_3) を3つのプロセス (P_1, P_2, P_3) が競合する場合を想定する。

ステップ 1：P_1 が R_1 を要求し、R_1 が P_1 に割り当てられる。
ステップ 2：P_2 が R_2 を要求し、R_2 が P_2 に割り当てられる。
ステップ 3：P_3 が R_3 を要求し、R_3 が P_3 に割り当てられる。
ステップ 4：P_2 が R_1 を要求し、待つ。
ステップ 5：P_3 が R_2 を要求し、待つ。

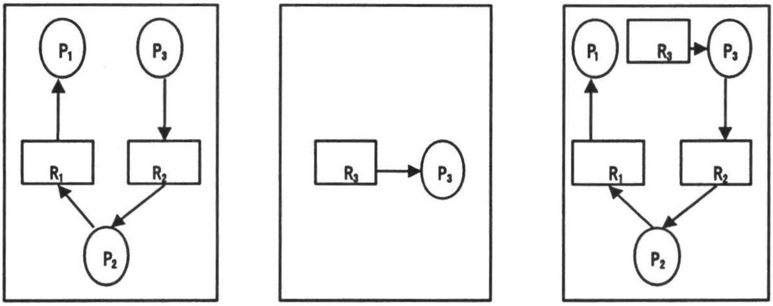
ステップ 6：P_1 が R_1 を解放し、R_1 が P_2 に割り当てられる。
ステップ 7：P_1 が R_3 を要求し、待つ。

連続的転送 (continuous transfer) の方法が適用されていると仮定すると、次のメッセージ系列が中央コーディネータに送信される。

m_1：サイト S_1 からエッジ (R_1, P_1) の追加
m_2：サイト S_1 からエッジ (R_2, P_2) の追加
m_3：サイト S_2 からエッジ (R_3, P_3) の追加
m_4：サイト S_1 からエッジ (P_2, R_1) の追加
m_5：サイト S_1 からエッジ (P_3, R_2) の追加
m_6：サイト S_1 からエッジ (R_1, P_1)、(P_2, R_1) の削除及びエッジ (R_1, P_2) の追加
m_7：サイト S_2 からエッジ (P_1, R_3) の追加

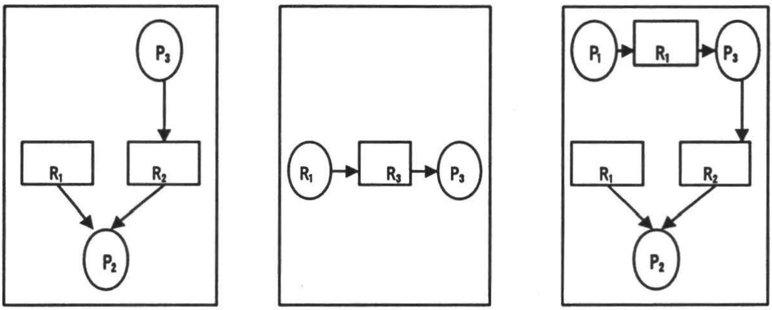
2 つのサイトのローカルコーディネータ、及び中央コーディネータによって維持されている資源割り当てグラフを図 6.16 に示す (説明を明確にするために、WFG の代わりに資源割り当てグラフを示す)。図 6.16(a) はステップ 5 の後、すなわちメッセージ m_5 が中央コーディネータによって受信された後のグラフを示す。また図 6.16(b) はメッセージ m_7 が中央コーディネータによって受信された後のグラフを示す。図 6.16(b) の中央コーディネータのグラフはサイクルをもっていないので、システムにはデッドロックが存在しないことを示している。しかし、サイト S_2 からのメッセージ m_7 がサイト S_1 からのメッセージ m_6 よりも前に中央コーディネータによって受信されたとする。この場合、中央コーディネータのシステムに関するビューは図 6.16(c) の資源割り当てグラフとなるであろう。それゆえ、中央コーディネータはデッドロックが発生したと間違って判断し、デッドロック回復のためのアクションを開始するであろう。上記の例は情報の連続的転送方法を用いる場合の幽霊デッドロックの検出の可能性を示しているが、他の情報転送方法でも、不完全又は遅れた情報のために、幽霊デッドロックが検出されることがありうる。

偽デッドロックの検出を避ける 1 つの方法は Lamport のアルゴリズムを用いて、各メッセージに全体でユニークなタイムスタンプをつけることである。上記の例では、サイト S_2 から中央コーディネータへのメッセージ m_7 はサイト S_1 からの要求によって引き起こされる (ステップ 7) ので、メッセージ m_7 は



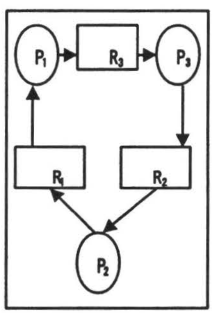
サイトS₁のローカルコーディネータの資源割り当てグラフ　　サイトS₂のローカルコーディネータの資源割り当てグラフ　　中央コーディネータによって維持されている資源割り当てグラフ

(a) ステップ5の後の資源割り当てグラフ



サイトS₁　　サイトS₂　　中央コーディネータ

(b) ステップ7の後の資源割り当てグラフ



中央コーディネータ

(c) もしメッセージm_7がメッセージm_6の前に中央コーディネータによって受信されるなら偽デッドロックが中央コーディネータの資源割り当てグラフにおいて現れる

図 6.16　集中型デッドロック検出方式におけるローカル及び全体資源割り当てグラフ

メッセージ m_6 よりも後のタイムスタンプをもつであろう。もし中央コーディネータが m_6 の前にメッセージ m_7 を受信し、偽デッドロックを検出するならば、デッドロックを解除するためのアクションをとる前に、検出されたデッドロックが本当のものか否か最初に確認する。確認のために、全体 WFG の更新のためのタイムスタンプ T よりも前のタイムスタンプをもつメッセージをサイトがもっていないかすべてのサイトに問い合わせるためのメッセージをブロードキャストする。このメッセージを受信すると、もしサイトが T よりも前のタイムスタンプをもつメッセージをもっているならば、直ちに中央コーディネータにそれを送信する、そうでなければ否定応答を単に送信する。すべてのサイトから応答を受信した後に、中央コーディネータは全体 WFG を更新 (もし更新メッセージがあれば) し、前に検出されたサイクルがまだ存在するならば、デッドロックが真のものであると結論して、回復アクションを起動する。上記の例では、ブロードキャストメッセージへの応答において、中央コーディネータはサイト S_1 からメッセージ m_6 を、そしてサイト S_2 から否定応答を受信するであろう。それゆえ、全体グラフの最後の更新の後に、中央コーディネータのシステムへのビューは図 6.16(c) のビューから図 6.16(b) のビューへと変わるであろう。したがって、デッドロック解決アクションは起動されないであろう。

デッドロック検出の階層的方法

典型的なアプリケーションにおいてほとんどの WFG サイクルの長さは短い。特に、実験による観測では、デッドロックサイクルの 90%は 2 つのプロセスによって起きている [Gray 他 1981]。それゆえ、集中型の方法は、中央コーディネータにローカル WFG を集めるのに必要な時間とメッセージオーバヘッドが存在するので、ほとんどのアプリケーションにとってあまり魅力的とはいえない。さらには、通信コストを最小にするため、地域的に分散しているシステムにおいて、デッドロックはサイクルに含まれるサイトにできるだけ近い位置のサイトによって検出されるべきである。しかしこれは集中型の方法では不可能である。階層的な方法は集中型のこれらの欠点を解決する。

階層的デッドロック検出方法はデッドロック検出の論理的な階層 (ツリー) を用いる。これらのデッドロック検出者はコントローラと呼ばれる。各コントローラは階層における自己の範囲のサイトを含むデッドロックにのみ責任がある。それゆえ、単一のサイトが全体 WFG を維持する集中型の方法とはちがって、階層的方法では、全体 WFG を維持するサイトは多くの異なるコントローラに分散している。各サイトは自己のローカルグラフを維持するローカルコン

トローラをもっている。

コントローラの階層を表すツリーにおいて、特定のコントローラによって維持されるべき WFG は次のルールによって決定される。

1. 階層ツリーのリーフを形成する各コントローラは1つのサイトのローカル WFG を維持する。

2. リーフでない各コントローラは、階層ツリーにおける直下の子の WFG の合併 (union) である WFG を維持する。

WFG においてサイクルを発見する最下位レベルのコントローラはデッドロックを検出し、解決するために必要なアクションをとる。それゆえ、サイクルを含む WFG は高位レベルのコントローラに手渡されることはないであろう。

図 6.17 に示される例を用いて方法を説明する。システムに4つのサイトと7つのコントローラが存在する。コントローラ A、B、C、及び D はそれぞれサイト S_1、S_2、S_3、及び S_4 のローカル WFG を維持する。これらはコントローラの階層ツリーのリーフを形成する。コントローラ A 及び B の親であるコントローラ E はコントローラ A 及び B の WFG の合併 WFG を維持する。同様に、コントローラ F はコントローラ C 及び D の WFG の合併 WFG を維持する。最後に、コントローラ G はコントローラ E 及び F の WFG の合併 WFG を維持する。

図から、サイト S_1 及び S_2 を含むデッドロックサイクル (P_1, P_3, P_2, P_1) はコントローラ E の WFG に反映されるが、サイト S_2、S_3、及び S_4 を含むデッドロックサイクル $(P_4, P_5, P_6, P_7, P_4)$ はコントローラ G の WFG にのみ反映される。これは、コントローラ G が最初の階層のコントローラであり、その配下に3つのサイト S_2、S_3、及び S_4 が含まれているからである。また、コントローラ G の WFG におけるデッドロックサイクル (P_1, P_3, P_2, P_1) はコントローラ E 及び F の WFG の合併 WFG に反映していることを示したが、これは実際には決して起きない。これは、コントローラ E がデッドロックを検出すると、コントローラ G に WFG をそのまま手渡す代わりに回復アクションを起動するからである。

デッドロック検出の完全分散方式

デッドロック検出の完全分散方式において、システムの各サイトはデッド

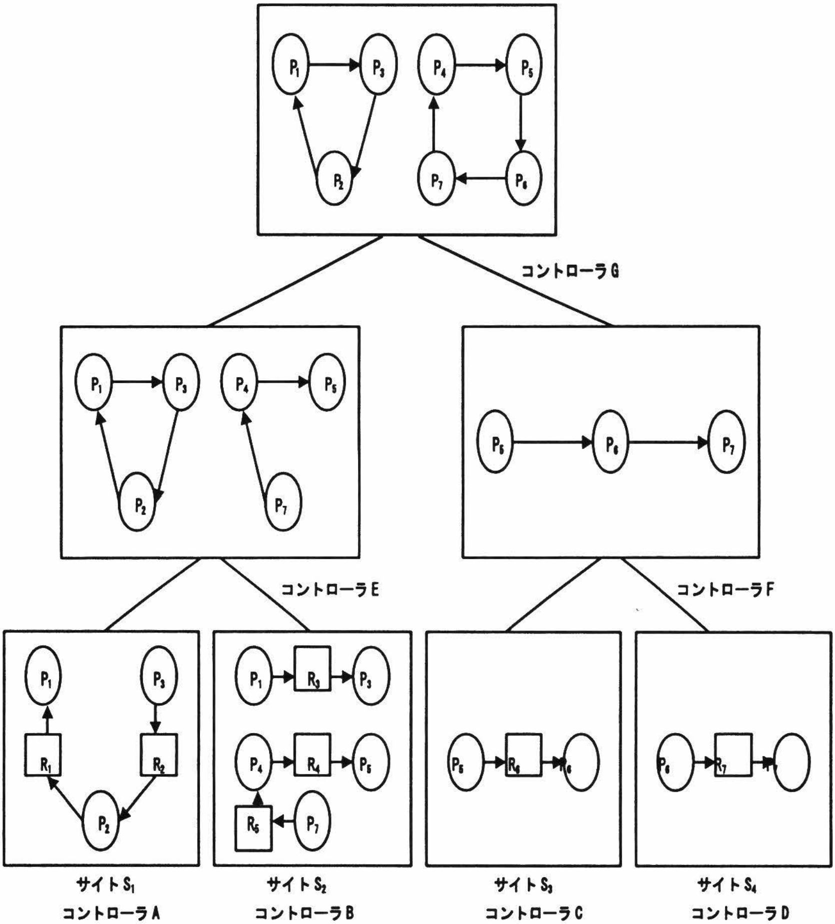


図 **6.17**　階層的デッドロック検出方法

ロック検出に平等の責任を分担する。この方式の幾つかのアルゴリズムのサーベイは [Knapp 1987, Singhal 1989] に見出すことができる。以下に 2 つのアルゴリズムを述べる。最初のものは WFG の構築に基づき、2 番目のものは探査ベース (probe-based) のアルゴリズムである。

デッドロック検出のための WFG に基づく分散アルゴリズム

以下の説明は [Silberschatz 及び Galvin 1994] によって示された完全分散デッドロック検出アルゴリズムに基づく。集中型及び階層型の方法と同様に、WFG に基づく分散アルゴリズムにおいて、各サイトは自己のローカル WFG を維持する。しかし、外部 (非ローカル) プロセスを含めて待っている状況をモデル化するため、WFG の若干の変更形が使用される。この修正された WFG では、追加ノード P_{ex} が各サイトのローカル WFG に追加され、このノードは次に述べる仕方で、対応するサイトの WFG に接続される。

1. もしプロセス P_i が他のプロセスによって保持されている他のサイトの資源を待っているなら、エッジ (P_i, P_{ex}) が追加される。

2. もし P_j がこのサイトのプロセスによって現在保持されている資源を待っている他のサイトのプロセスであるならば、エッジ (P_{ex}, P_j) が追加される。

この修正 WFG の構築を説明するために、図 6.18 の例を考察する。この例では、2 つのサイトがあり、各サイトのローカル WFG を図 6.18(a) に示す。ノード P_{ex} の追加後の 2 つのサイトの修正 WFG を図 6.18(b) に示す。ノード P_{ex} を含むエッジの説明を以下に述べる。

1. サイト S_1 の WFG において、プロセス P_1 はプロセス P_3 によって保持されているサイト S_2 の資源を待っているので、エッジ (P_1, P_{ex}) が追加される。また、サイト S_1 のプロセス P_2 によって現在保持されている資源の獲得を待っているプロセス P_3 はサイト S_2 のプロセスであるので、エッジ (P_{ex}, P_3) が追加される。

2. サイト S_2 の WFG において、プロセス P_2 によって保持されているサイト S_1 の資源をプロセス P_3 が待っているので、エッジ (P_3, P_{ex}) が追加される。また、プロセス P_1 はサイト S_1 のプロセスであり、サイト S_2 のプロセス P_3 に

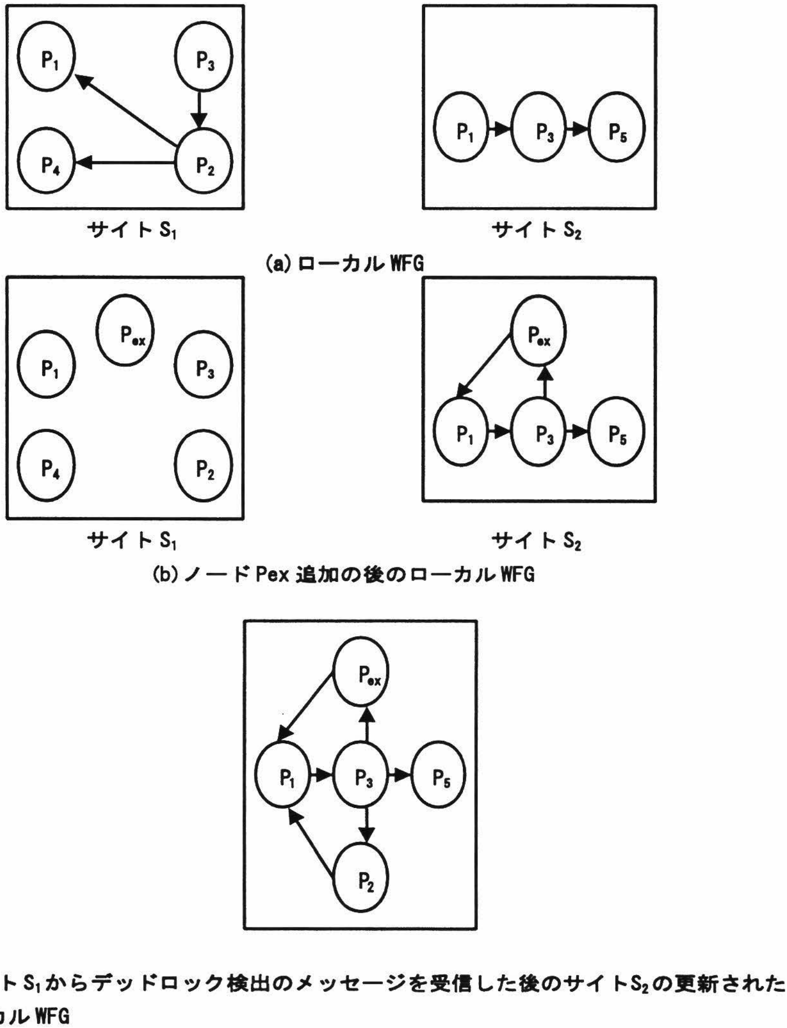


図 6.18　完全分散デッドロック検出アルゴリズム

よって現在保持されている資源を獲得するために待っているので、エッジ (P_{ex}, P_1) が追加される。

それから、次の方法でデッドロック検出のために、修正 WFG が使用される。もし、ローカル WFG がノード P_{ex} を含まないサイクルをもっているならば、そのサイトのローカルプロセスを含むデッドロックが発生したことになる。このようなデッドロックは他のサイトと相談する必要なくローカルに解決されうる。

他方、もしローカル WFG がノード P_{ex} を含むサイクルをもっているならば、複数サイトのプロセスを含む分散デッドロックの可能性がある。分散デッドロックを確認するために、分散デッドロック検出アルゴリズムは、その WFG にノード P_{ex} を含むサイクルをもつサイトによって起動される。アルゴリズムは次に述べるように動作する。

ノード P_{ex} を含むサイクルがサイト S_i の WFG において検出されるとする。このサイクルは次の形式になる。

$$(P_{ex}, P_i, P_j, \ldots\ldots, P_k, P_{ex})$$

ここで、プロセス P_k は他のどれかのサイト (S_j としよう) に属する外部資源を待っていることを意味する。それゆえ、サイト S_i はサイト S_j にデッドロック検出メッセージを送信する。このメッセージはサイト S_i の完全な WFG を含まないが、サイクルを形成する WFG の一部を含む。例えば、図 6.18 の例において、もしサイト S_1 が最初にサイクルを検出するならば、P_1 はサイト S_2 における資源を待っているので、(P_{ex}, P_3, P_2, P_1, P_{ex}) のようなメッセージをサイト S_2 に送信する。

メッセージを受信すると、サイト S_j は、ノード P_{ex} を含まないサイクルのエッジをその WFG に追加することにより、ローカル WFG を更新する。すなわち、この例ではエッジ (P_3, P_2) 及び (P_2, P_1) はサイト S_2 のローカル WFG に追加され、その結果図 6.18(c) の新しい WFG となる。

もしサイト S_j の新しく構築された WFG が P_{ex} を含まないサイクルをもつならば、デッドロックは存在し、適切な回復手続きが起動されなければならない。例えば、この例では、新しく構築されたサイト S_2 の WFG は、ノード P_{ex} を含まないサイクル (P_1, P_3, P_2, P_1) をもつ。したがって、この例では、システムはデッドロック状態になる。

他方、ノード P_{ex} を含むサイクルがサイト S_j の新しく構築された WFG の

中に存在すると、S_j は適切なサイト (S_k としよう) にデッドロック検出メッセージを送信する。そしてサイト S_k によって全体の手続きが繰り返される。この方法で、有限数のデッドロック検出メッセージが 1 つのサイトから別のサイトに転送した後に、デッドロックが検出されるか又はデッドロック検出のための計算は停止するかのどちらかである。

上記のアルゴリズムに関する問題は、同じプロセスを含むデッドロックを検出するために、2 つのサイトがデッドロック検出アルゴリズムを独立に起動するかもしれないという点である。例えば、図 6.18 のこの例では、サイト S_1 及び S_2 はそれぞれのローカル WFG においてほとんど同時にサイクル (P_{ex}, P_3, P_2, P_1, P_{ex}) 及び (P_{ex}, P_1, P_3, P_{ex}) を検出するかもしれない。そして両サイトとも他のサイトにデッドロック検出メッセージを送信するかもしれない。その結果、両サイトともそれぞれの WFG を更新し、サイクルを探索するであろう。デッドロックを検出した後に、両サイトは回復手続きを起動するかもしれない、その結果デッドロックを解決するのに実際に必要な数以上にプロセスを強制終了させるかもしれない。さらに、この問題はまた、不必要なメッセージ転送の余計なオーバヘッド及び 2 つのサイトで行われるデッドロック検出処理の重複をもたらす。

上記の問題を解決する 1 つの方法は、各プロセス P_i に固有識別子［ID(P_i) とする］を割り付けることである。あるサイトのローカル WFG において (P_{ex}, P_i, P_j, ... , P_k, P_{ex}) の形式のサイクルが検出されると、このサイトは次の条件のときのみ、適切なサイトにデッドロック検出メッセージを送信することによりデッドロック検出アルゴリズムを起動する。

$$ID(P_k) < ID(P_i)$$

そうでなければ、このサイトは何も行わず、デッドロック検出アルゴリズムを起動するというジョブは他のサイトにまかせる。

図 6.18 の例に、修正アルゴリズムを適用する。

$$ID(P_i) < ID(P_2) < ID(P_3) < ID(P_4) < ID(P5)$$

とする。

両サイト S_1 及び S_2 がそれぞれのローカル WFG でサイクル (P_{ex}, P_3, P_2, P_1, P_{ex}) 及び (P_{ex}, P_1, P_3, P_{ex}) をほとんど同時に検出するとする。$ID(P_1) < ID(P_3)$ であるので、サイト S_1 がデッドロック検出アルゴリズムを

起動する。他方、$ID(P_3) > ID(P_1)$ であるので、サイト S_2 はそのローカル WFG にサイクルがあるにも拘わらず何も行わない。サイト S_2 がサイト S_1 によって送信されたデッドロック検出メッセージを受信すると、そのローカル WFG を更新し、更新された WFG においてサイクルを探索する。そして、グラフにおいてサイクル (P_1, P_3, P_2, P_1) を検出し、デッドロック回復手続きを起動する。

探査ベースのデッドロック検出分散アルゴリズム

以下に述べる探査ベースのデッドロック検出分散アルゴリズムは Chandy 他 [1983] によって提案された。そして Chandy-Misra-Hass(又は CMH) アルゴリズムとして知られている。これは今日まで、分散システムにおける全体デッドロックの検出のための最良のアルゴリズムと考えられている。このアルゴリズムでは、あるプロセスが同時に複数の資源を要求できる。

このアルゴリズムは概念的には単純で、次のように動作する。資源 (1 つ又は複数) を要求するプロセスが要求した資源を獲得できずタイムアウトになる場合に、プロセスは特別の**探査 (probe)** メッセージを生成し、資源を保持しているプロセス (1 つ又は複数) にメッセージを送信する。探査メッセージは次のフィールドをもっている (システムにおける各プロセスはユニークな識別子を割り付けられているとする)。

1. ブロックしたプロセスの識別子
2. このメッセージを送信しているプロセスの識別子
3. このメッセージが送信される相手のプロセスの識別子

探査メッセージを受信すると、受信者は自身が資源 (1 つ又は複数) を待っているか否か検査する。資源を待っていなければ、受信者はこのメッセージを送信したプロセスによって要求された資源を使用していることを意味する。この場合、受信者は探査メッセージを無視するのみである。他方、もし受信者が資源を待っているならば、待っている資源 (1 つ又は複数) を保持しているプロセス (1 つ又は複数) に、探査メッセージを転送する。しかし、探査メッセージを転送する前に、受信者は次の方法でフィールドを修正する。

1. 最初のフィールドはそのままとする。
2. 受信者は第 2 のフィールドを自己の識別子に変更する。

3. 第 3 のフィールドを、このメッセージの新しい受信者となるプロセスの識別子に変更する。

探査メッセージのすべての新しい受信者はこの手続きを繰り返す。もし探査メッセージが元の送信者 (メッセージの第 1 フィールドの識別子のプロセス) に戻る場合、サイクルが存在し、システムはデッドロック状態にある。

図 6.19 に示す単純な例を用いてアルゴリズムを説明する。この図は図 6.18(a) ノード状態と同じ状態を表すが、表現が若干異なる。プロセス P_3 によって保持されている資源を要求するときプロセス P_1 がブロックするものとする。それゆえ、P_1 は探査メッセージ (P_1, P_1, P_3) を生成し P_3 に送信する。P_3 がこのメッセージを受信すると、自身がプロセス P_2 及び P_5 によってブロックしていることを知る。それゆえ、P_3 は探査メッセージ (P_1, P_3, P_2) を P_2 に、探査メッセージ (P_1, P_3, P_5) を P_5 に転送する。P_5 が探査メッセージを受信すると、P_5 は他のどのプロセスによってもブロックされていないので、メッセージを無視する。しかし、P_2 が探査メッセージを受信すると、自身は P_1 及び P_4 によってブロックされていることを知る。それゆえ、P_2 は探査メッセージ (P_1, P_2, P_1) を P_1 に、探査メッセージ (P_1, P_2, P_4) を P_4 に転送する。探査メッセージが元の送信者 (P_1) に戻るので、サイクルが存在し、システムはデッドロック状態にある。

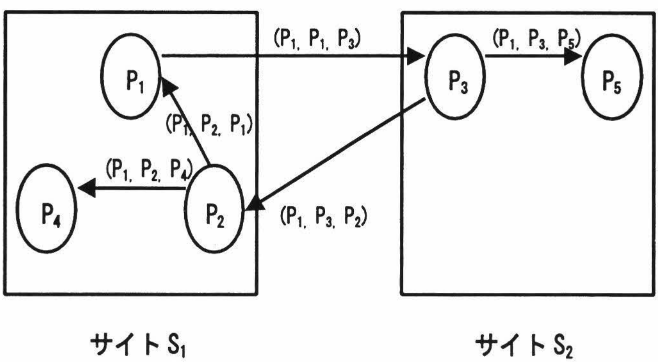


図 6.19　CMH 分散デッドロック検出アルゴリズムを示す例

CMH アルゴリズムは一般的であり、以下に述べる魅力ある特徴のゆえに、このアルゴリズムの変形がほとんどの分散ロック方式に使用されている。

1. 各メッセージは固定長であり、必要な計算量も少ないので、このアルゴリズムは実装が容易である。
2. このアルゴリズムのオーバヘッドは比較的少ない。
3. グラフの構築や情報収集が不要である。
4. このアルゴリズムでは偽デッドロックは検出されない。
5. プロセス間に特別の構造を必要としない。

デッドロックからの回復方法

デッドロックをシステムとして扱うのに、検出及び回復方策を使用することを選ぶとすると、デッドロックを検出するだけでは不十分である。システムは検出されたデッドロックから回復する何らかの方法を備えるべきである。デッドロックからの回復方法として次の方法の1つが使用される。

■オペレータの介入を求める。
■プロセスの強制終了
■プロセスのロールバック

オペレータの介入を要請する方法最も単純な方法は、オペレータにデッドロックが発生したことを知らせ、オペレータがデッドロックを手動で扱うようにすることである。システムはデッドロックに関連するプロセスのリストをオペレータに提供して判断を支援する。

この方法は現代システムにおいて余り使用に適していない。オペレータがコンソールからシステムの円滑な動作を常時監視するという考えは最近なくなりつつある。さらに、この方法は集中型システムではうまく動作するかもしれないが、分散システム環境では動作しない。その理由は、デッドロックに複数のサイトが関与していても、どのサイトに知らせるか判断できないからである。もしデッドロックに関係するプロセスの存在するすべてのサイトが知らされるならば、各サイトのオペレータは独立に回復のためのアクションをとるであろう。他方、もしただ1つのサイトのオペレータが知らされると、そのオペレータは回復アクションの処理を行っている間に、自己のサイトのプロセスを優先させるかもしれない。さらには、1つのサイトのオペレータは回復アクションをとるために、別のサイトのプロセスと情報交換する権利をもっていないであろう。それゆえ、分散システムでは通常、デッドロックからシステムが自動的

に回復する以下に述べる別の方法を使用する。

プロセスの強制終了デッドロックから自動的に回復する最も単純な方法は、1 つ以上のプロセスを強制終了させ、それらのプロセスによって保持されていた資源を再度要求することである。この考えに基づくデッドロック回復アルゴリズムは資源要求を分析し、デッドロックサイクルに含まれるプロセスの相互依存性を分析し、そしてもし強制終了させればサイクルを切断できるプロセスの集合を選択する。

プロセスのロールバックプロセスを強制終了させることは、プロセスを最初から再開始させることを要することになり、特にプロセスが相当長時間既に動作した後には、非効率なものとなる。デッドロック状態から脱するには、強制終了させることを選択したプロセスから必要な資源を返却させることで十分である。また、プロセスから資源を返却させるためには、プロセスが資源を割り当てられていない時点までプロセスをロールバックすることで十分であることに注意されたい。ロールバックの方法はこの考えに基づく。

この方法では、プロセスは周期的にチェックポイントが行われる。すなわち、プロセスの状態 (メモリのイメージ及び保持している資源のリスト) は規則的な時間間隔でファイルに書かれる。それゆえ、ファイルはプロセスの履歴を維持し、もし要求されるなら、そのプロセスはどのチェックポイントからでも再開始できる。チェックポイントが検出されると、強制終了させるプロセスの集合を選択するために、プロセス強制終了方法において述べた方法が使用される。しかし、今回は、選択されたプロセスを完全にロールバック (すなわち強制終了) させる代わりに、デッドロックを脱するのに必要な時点まで、プロセスをロールバックさせる。すなわち、各選択されたプロセスを、必要な資源を返却する時点のチェックポイントまでロールバックさせる。

ロールバック方法はプロセス強制終了方法よりも効率的に見えるが、すべてのプロセスの周期的なチェックポイントをとるというオーバヘッドがあるので、必ずしも効率的ではない。もしデッドロックがシステムに希にしか起きないならば、プロセス強制終了方法の方が効率的である。

デッドロックからの回復における問題

回復アクションにおいて 2 つの重要な問題は犠牲者の選択及びトランザクション機構の使用である。以下にそれらを述べる。

犠牲者の選択

上記の回復方法のどの方法においても、デッドロックは、1つ以上のプロセスを強制終了又はロールバックさせせることによって脱する。これらのプロセスは犠牲者と呼ばれる。オペレータ介入の方法でも、回復は1つ以上の犠牲者を強制終了させることを含んでいる。それゆえ、どの回復手続きにおいても、重要な問題は犠牲者の選択である。犠牲者の選択は2つの要素に基づいている。

1. **回復コストの最小化:** コスト要素は、プロセスの強制終了又はロールバックによって生じる回復コストが最小となるようなプロセスを犠牲者として選択すべきことを示している。不幸にも、汎用的なコスト関数を定義することは不可能であるので、各システムは犠牲者を選択するための独自のコスト関数を定義しなければならない。この目的のために考慮すべき要素は、(a) プロセスの優先度、(b) プロセスの特性、例えば対話型かバッチ型か、及び悪影響なく再動作する可能性、(c) プロセスによって保持される資源の数とタイプ、(d) 既に受けたサービスの長さ、及びプロセスによってさらに必要な予測されるサービスの長さ、及び (e) 影響を受けるプロセスの総数の要素である。

2. **飢餓の予防:** もしシステムが回復コストの最小化のみを狙っているならば、同じプロセス (多分、優先度が非常に低い) が犠牲者として繰り返し選択されて、そのプロセスは永久に完了しないかもしれない。飢餓として知られている、この状況は、実際的なシステムでは、何等かの方法で予防されなければならない。この問題を扱う1つの方法は犠牲にされた時点でプロセスの優先度を上げることである。別の方法は、コスト関数のパラメータとして、あるプロセスが犠牲にされた回数を含ませることである。

トランザクション機構の使用

デッドロックからの回復のためにあるプロセスが強制終了又はロールバックされた後に、そのプロセスは再動作されなければならない。しかし、プロセスの再動作は必ずしも安全ではない、特にプロセスによって既に行われた操作が非繰り返し等価 (nonidempotent) の場合、安全ではない。例えば、銀行口座にある金額を加算することにより、あるプロセスが銀行口座の金額を更新したならば、プロセスの再実行は同じ金額を口座に加算することになり、口座の残高は不正な状態となる。それゆえ、トランザクション機構の使用 (すべてか皆無

(all or no effect) を保証する) は、デッドロックを扱うのに検出及び回復の方法を選択するシステムにとって、ほとんど不可避となる。しかし、トランザクション機構は、悪影響無しに再動作可能なプロセスのためには使用する必要がない。例えば、コンパイル処理において、ソースファイルを読み取りオブジェクトファイルを作成するのが処理のすべてであるので、コンパイルプロセスの再動作は悪影響を及ぼさない。

6.6　選任アルゴリズム

分散アルゴリズムの幾つかは、全体システムにおいて 1 つのコーディネータプロセスを必要とする。コーディネータプロセスは、システムにおいて他のプロセスの円滑な動作のために必要なある種の調整行為を行う。本章において、このようなコーディネータプロセスの 2 つの例は、相互排他のための集中型アルゴリズムにおけるコーディネータ、及び集中型デッドロック検出アルゴリズムにおける集中型コーディネータである。システムにおけるすべてのプロセスはコーディネータと情報交換しなければならないので、すべてはだれがコーディネータかについて全員一致で合意していなければならない。さらに、コーディネータプロセスが存在するサイトの故障によりコーディネータプロセスが故障すると、故障したコーディネータの仕事を引き継ぐために、新しいコーディネータが選任されなければならない。**選任アルゴリズム (election algorithm)** は、システムのすべてのプロセスに 1 つのコーディネータがいつでも存在するように、現在動作中のプロセスからコーディネータプロセスを選任することを意味する。

選任アルゴリズムは次の仮定に基づく。

1. システムの各プロセスはユニーク優先度番号をもっている。
2. 選任が行われるときはいつも、現在動作中のプロセスで最高の優先度番号をもつプロセスがコーディネータとして選任される。
3. 回復時、故障プロセスが動作中プロセスの集合に再度入るために適切なアクションをとることができる。

それゆえ、起動時はいつも、選任アルゴリズムは現在動作中のプロセスの中で、どれが最高の優先度番号をもつかを見つけ、他の動作中プロセスすべてにこれを知らせる。異なる選任アルゴリズムは、この処理の方法において異な

る。以下に2つの選任アルゴリズムを述べる。他の選任アルゴリズムに興味のある読者は [Tel 1994] を参照されたい。簡単のために、両者のアルゴリズムの記述において、分散システムの各ノードにただ1つのプロセスが存在すると仮定する。

6.6.1 ブリーアルゴリズム

このブリーアルゴリズム (**bully algorithm**) は Garcia-Molina [1982] によって提案された。このアルゴリズムにおいて、各プロセスはシステムの他のプロセスの優先度番号を知っていることを仮定している。アルゴリズムは次のように動作する。

あるプロセス (P_i とする) はコーディネータに要求 (request) がメッセージを送信し、一定時間以内に応答を受信しないと、コーディネータは故障したと判断する。それから、そのプロセスよりも高い優先度番号をもつ各プロセスに選任 (election) メッセージを送信することにより、選任を開始する。もし P_i が一定時間以内に選任メッセージに対する応答を受信しないならば、現在動作中プロセスの中で自分が最高の優先度番号をもっているとみなす。それゆえ、コーディネータの仕事を引き受け、そのプロセスよりも低い優先度番号をもつすべてのプロセスにメッセージ (コーディネータ (coordinator) メッセージと呼ぼう) を送信する。そのメッセージはこれからコーディネータになるプロセスからのメッセージを知らせる。他方、もし P_i がその選任メッセージの応答を受信すると、これは、より高い優先度番号をもつ他のプロセスが生きていることを意味する。それゆえ、P_i はそれ以上のアクションを行わず、P_i が発動した選任の最終的な結果 (新しいコーディネータからのコーディネータメッセージ) の受信を待つ。

あるプロセス (P_j とする) が選任メッセージ (明らかにそれ自身よりも低い優先度番号をもつプロセスから) を受信すると、存在することと選任行為を引き継ぐことを送信者に知らせる応答メッセージ生存 (alive) メッセージと呼ぼう) を送信する。P_j は既に選任を行っている場合を除き、選任行為を行う。このようにして、選任行為は、現在動作中プロセスの間で最高の優先度番号をもつプロセスに徐々に移動していき、実質的に選任を勝ち取り新しいコーディネータになる。

回復アクションの一部として、この方法は故障を見つけたプロセス (P_k とする) が回復のための選任を発動しなければならない。もし現在のコーディネー

タの優先度番号が P_k の番号よりも高い場合には、現在のコーディネータは P_k によって発動された選任を勝ち取り、コーディネータであり続けるであろう。他方、もし P_k の優先度番号が現在のコーディネータの番号よりも高い場合には、選任メッセージに対する応答を受信しないであろう。そして選任を勝ち取り、現在動作中のコーディネータからコーディネータの仕事を引き継ぐ。それゆえ、最高の優先度番号をもつ動作中プロセスがいつも選任を勝ち取る。そこで、このアルゴリズムは「ブリー」アルゴリズムと呼ばれる。もし最高の優先度番号をもつプロセスが故障の後に回復するならば、システムにおいて自己の優先度番号が一番高いと優先度番号リストから知っているので、選任を発動しない。それゆえ、回復時、他のすべてのプロセスに単にコーディネータメッセージを送信し、現在のコーディネータにとって代わる。

例を用いてこのアルゴリズムの動作を説明する。システムは 5 つのプロセス P_1、P_2、P_3、P_4、及び P_5 からなり、それぞれの優先度番号が 1、2、3、4、及び 5 であるとする。また、特定の瞬間の時点において、P_2 が故障し、P_1、P_3、P_4、及び P_5 が動作中の状態にシステムがあるとする。この状態からスタートし、システム状態の変化とともにブリーアルゴリズムの機能を以下に説明する。

1. 明らかに、開始時点の状態で P_5 がコーディネータである。

2. P_5 が故障したとする。

3. プロセス P_3 は P_5 に要求メッセージを送信し、一定時間以内に応答を受信しない。

4. プロセス P_3 は P_5 が故障したとみなし、P_4 と P_5(選任メッセージはより高い優先度番号をもつプロセスにのみ送信される) に選任メッセージを送信することによって選任を発動する。

5. P_4 が P_3 の選任メッセージを受信すると、P_3 に生存メッセージを送信し、P_4 が生存していること及び選任行為を引き継ぐことを知らせる。プロセス P_5 はダウンしているので P_3 からの選任メッセージに応答できない。

6. P_4 は P_5 に選任メッセージを送信することにより選任を開催することを知らせる。

7. プロセス P_5 はダウンしているので、P_4 の選任メッセージに応答しない。それゆえ、P_4 は選任を勝ち取り、コーディネータメッセージを P_1、P_2、及び P_3 に送信し、今から P_4 が新しいコーディネータであることを知らせる。明らかに、このメッセージは、P_2 がダウンしているので P_2 によって受信されない。

8. P_2 が故障から回復し、選任メッセージを P_3、P_4、及び P_5 に送信するこ

とにより選任を発動する。P_2 の優先度番号は P_4(現在のコーディネータ) の番号よりも小さいので、P_4 は P_2 により発動された選任を勝ち取り、コーディネータであることを継続する。

9. 最後に、P_5 が故障から回復するとする。P_5 は最高の優先度番号をもっているので、P_5 はコーディネータメッセージを P_1、P_2、P_3、及び P_4 に送信するのみで、新しいコーディネータとなる。

6.6.2 リングアルゴリズム

次のアルゴリズムは、[Tanenbaum 1995, Silberschatz 及び Galvin 1994] において示されたリングベースの選任アルゴリズム (ring-based election algorithm) に基づく。このアルゴリズムでは、すべてのプロセスが論理的リングに組織化されていると仮定する。選任アルゴリズムに関するすべてのメッセージは常に一方向 (時計回り又は反時計回り) に渡っていくという意味で、リングは単方向である。システムにおける各プロセスはリングの構造を知っているので、リング上をメッセージが循環する間に、もし送信者の次のプロセスがダウンしていると、送信者は動作中のプロセスが存在するところまで、次々と後続者をスキップすることができる。アルゴリズムは次のように動作する。

プロセス (P_j としよう) が現在のコーディネータに要求メッセージを送信し、一定時間以内に応答を受信しないとき、コーディネータが故障したとみなす。それゆえ、このプロセスは次の後続者 (実際は現在動作中の最初の後続者) に選任メッセージを送信することにより選任を発動する。選任メッセージを受信すると、後続者は自己の優先度番号をメッセージに添付し、リング上の次の動作中メンバーに転送する。このメンバーは自己の優先度番号をメッセージに添付し、その後続者に転送する。このようにして、選任メッセージは 1 つの動作中のプロセスから別のプロセスへと、リング上を循環し、実際にプロセス P_i に戻ってくる。プロセス P_i は、メッセージの中に保持された優先度番号のリストにおいて、最初の優先度番号が自己の優先度番号であることを確認することによって、そのメッセージが自己の選任メッセージであることを認識する。

プロセス P_i が自己の選任メッセージを受信すると、そのメッセージは現在動作中のすべてのプロセスの優先度番号のリストを含んでいる。それゆえ、リストの中のプロセスから、P_i は最高優先度番号をもつプロセスを新しいコーディネータとして選任する。それから、コーディネータメッセージをリング上に循環させ、どのプロセスが新しいコーディネータであるのかを、動作中の他のプ

ロセスに知らせる。リングを一巡して、コーディネータメッセージがプロセス P_i に戻ってくると、プロセス P_i によって取り除かれる。この時点で、すべての動作中プロセスは、どのプロセスが現在のコーディネータであるかを知る。

プロセス (P_j としよう) が故障から回復すると、質問 (inquiry) メッセージを生成し、次の後続者に送信する。このメッセージはプロセス P_j の識別子を含んでいる。もし後続者が現在のコーディネータでなければ、単に質問メッセージを次の後続者に転送する。このようにして、質問メッセージは現在のコーディネータに到達するまでリングにそって移動していく。質問メッセージを受信すると、現在のコーディネータは、自己が現在のコーディネータであることを知らせる応答を、プロセス P_j に送信する。

このアルゴリズムにおいて、2 つ以上のプロセスがほとんど同時にコーディネータが故障したことを発見し、それぞれがリング上に選任メッセージを循環させるかもしれない。この結果はネットワークのバンド幅を多少浪費するが、問題は生じない。選任を発動した各プロセスは、動作中プロセスの同じリストを受信し、それらのすべては新しいコーディネータとして同じプロセスを選択するであろう。

6.6.3　2 つの選任アルゴリズムの検討

ブリーアルゴリズムにおいて、最低の優先度番号をもつプロセスがコーディネータの故障を検出し選任を発動する場合で、システムに n 個のプロセスがある場合に、発動ごとに選任が次々となされて、全体で $n-2$ の選任が行われる。すなわち、動作中の最高優先度番号をもつプロセス及び故障したコーディネータプロセスを除くすべてのプロセスが自己より高い優先度番号をもつすべてのプロセスに対しメッセージを送信することにより選任を発動する。したがって、最悪の場合、ブリーアルゴリズムは $O(n^2)$ 個のメッセージを必要とする。しかし、故障したコーディネータの優先度番号よりも 1 小さい優先度番号をもつプロセスがコーディネータの故障を検出する場合は、そのプロセスは自己をコーディネータに直ちに選任し、$n-2$ 個のコーディネータメッセージを送信する。したがって、最善の場合、ブリーアルゴリズムは $n-2$ 個のメッセージを要するのみである。

他方、リングアルゴリズムにおいて、どのプロセスがコーディネータの故障を検出し、選任を発動するかに拘わらず、選任は常に $2(n-1)$ 個のメッセージを必要とする (コーディネータのみが故障した場合を想定する)。$n-1$ 個の

メッセージは選任メッセージの一巡のために必要であり、別の $n-1$ 個のメッセージはコーディネータメッセージの一巡のために必要である。

次に、プロセスの回復に含まれる複雑性について考察する。ブリーアルゴリズムにおいて、故障したプロセスは回復時に選任を発動しなければならない。それゆえ、回復アクションを発動するプロセスの優先度番号に依存して、ブリーアルゴリズムは最悪の場合、$O(n^2)$ 個のメッセージを必要とし、最善の場合 $n-1$ 個のメッセージを必要とする。他方、リングアルゴリズムにおいて、故障したプロセスは回復時に選任を発動しないで、現在のコーディネータを探索するのみである。したがって、リングアルゴリズムは、回復アクションに対し平均 $n/2$ 個のみのメッセージを必要とする。

結論として、ブリーアルゴリズムに比較して、リングアルゴリズムはより効率的であり、実装もより容易である。

6.7　まとめ

システム資源を複数の並行プロセスで共有することは、協調的であるか又は競合的である。協調的及び競合的共有は共に、適正な相互動作になることを保証するために、相互動作についてのルールを必要とする。適正な相互動作を強制するためのルールは同期機構という形式で実装される。この章では、分散システムにおける同期問題を概観し、これらの問題を扱う機構を説明した。

幾つかの分散アプリケーションの適正な機能のために、分散システムの異なるノードのクロックは、外界 (物理時計) と同期しているのと同様に相互にも同期していなければならない。分散システムにおいて使用されるクロック同期アルゴリズムは大きく、集中型及び分散型の 2 つのタイプに分類される。集中型の方法では、1 つの時間サーバノードが存在し、アルゴリズムの目標は他のすべてのノードのクロックを時間サーバノードのクロック時間に同期させることである。分散型の方法では、クロック同期は、システムの様々なノードのクロックを全体で平均又はローカルで平均することにより達成される。

Lamport は、ほとんどのアプリケーションにとって、クロック同期は不要であり、分散システムにおいて発生するすべての事象が、観測される振る舞いと整合しているように全体で順序つけられていることを保証すれば十分であることを見出した。それゆえに、彼は先行生起関係を定義し、 先行生起関係に基づき事象の順序つけを行うための論理クロックの概念を導入した。しかし、先行生起関係はシステムのすべての事象の集合の半順序を与えるのみである。それ

ゆえ、すべてのシステム事象集合の全順序の順序つけを行うために、Lamport はプロセスの任意の全順序の使用を提案した。

システムにおいて、1つのプロセスによって排他的にアクセスすることを保証しなければならない幾つかの資源がある。このアクセスの排他性はプロセス間の相互排他と呼ばれ、資源を共有するために排他的アクセスを必要とするプログラムのセクション(領域)は臨界領域と言われる。分散システムにおける相互排他を実装するためのアルゴリズムに使用される3つの方法がある。それらは、集中型、分散型、トークンパッシング方式である。集中型の方法において、システムの1つのプロセスがコーディネータとして選任される。コーディネータは臨界領域に入ることを調整する。分散型の方法では、同じ臨界領域に入ろうとするすべてのプロセスは、次に臨界領域に入るプロセスの決定に至るまで、互いに協調する。トークンパッシング方式では、相互排他は、システムの中のプロセスの間を循環する1つのトークンを使用することにより達成される。

デッドロックはプロセスの集合が永久ブロックになっている状態である。この集合の各プロセスは集合内の別のプロセスのみが起こしうる事象を待っている。基本的に、分散システムにおけるデッドロックは集中システムにおけるデッドロックに類似している。しかし、分散システムにおけるデッドロックの扱いは集中システムにおけるよりも一層複雑である。複雑な理由は、資源、プロセス、及び他の適切な情報はシステムの異なるノードに分散しているからである。

デッドロックを扱うのに3つの一般によく使用される方策は、回避、予防、及び検出／回復である。デッドロック回避方法は、デッドロックになるような資源の割り当てを避けるために、システムの将来状態を予測することが必要であり、そのためには資源がプロセスによってどのように使用されるかという事前の知識を必要とする。デッドロック予防は、デッドロックが起きないようにシステムを注意深く設計することによる。検出／回復方法では、デッドロックは起きることが許容されており、システムによって検出され、それから回復される。3つの方法のうち、検出／回復方法が分散システムにおけるデッドロックを扱うための推奨方法である。分散システムにおいてデッドロック検出に3つの方法が使用される。それらは集中型、階層型、及び完全分散型である。検出されたデッドロックからの回復のために、システムは次のうちの1つの方法を使用するであろう。オペレータの介入を要請する方法、プロセスの強制終了、プロセスのロールバック。

幾つかの分散アルゴリズムはシステム全体に1つのコーディネータプロセスが存在することを必要とする。選任アルゴリズムは、現在動作中プロセスの間

から1つのコーディネータプロセスを選任することを意味する。本章で述べた2つの選任アルゴリズムはブリーアルゴリズム及びリングアルゴリズムである。

演習問題

6.1 与えられた一式のクロックが同期しているか否かを決定するアルゴリズムのための擬似コードを作成せよ。そのアルゴリズムにはどのような入力が必要であるか。

6.2 クロック同期の問題で、集中システムと分散システムとではどのように異なるか。

6.3 分散システムにおいて、すべてのノードのクロックを一度同期するのみでは不十分であり、周期的な再同期が必要な理由を述べよ。また周期的再同期の周期をどのように定めるか。

6.4 分散システムに3つのノード N_1, N_2, N_3 があり、それぞれが自己のクロックを持っている。ノード N_1, N_2, N_3 のクロックは1ミリ秒ごとにそれぞれ800、810、795回のティックを刻む。システムは外部同期機構を使用し、すべての3つのノードは外部から30秒ごとに実時間を受信し、それぞれのクロックを修正する。このシステムで発生する最大クロックスキューはどれだけか。

6.5 分散システムにおいてクロックの内部同期及び外部同期の区別をせよ。外部同期クロックは内部同期もなされている、しかしその逆は真ではない。その理由を説明せよ。

6.6 コンピュータシステムのクロック同期において重要な事項は、時間は決して逆戻りできないということである。この事がなぜ重要であるかを示す例を2つ挙げよ。これに関して、速いクロックがどのように調整されるか。

6.7 分散システムにおいて、2つのノード間に予測不可能な伝搬時間の変動が存在するかもしれない。その理由を述べよ。この問題がどのように分散システムにおけるクロックの同期処理を困難にしているか。この問題を扱うクロック同期アルゴリズムに使用できる2つの方法を挙げよ。

6.8 ほとんどの分散アプリケーションにとって、すべてのプロセス間で正確に時間がどうであるかに合意することが問題でなく、むしろ事象の生起順序について合意することが問題であることを示す2つの例を挙げよ。

6.9 図6.20の空間-時間図を用いて、先行生起関係に従って並行事象のすべての対を挙げよ。

6.10 図6.20の空間-時間図に、事象 e_5、e_6、及び e_7 に並行なメッセージ送

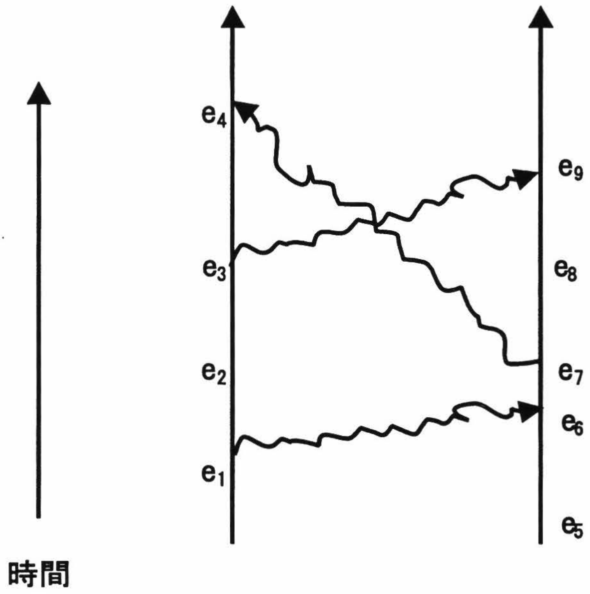


図 **6.20**　空間-時間図

信事象を追加せよ。また、事象 $e1$、e_2、及び e_3 に並行な非メッセージ送信事象を追加せよ。

6.11 分散システムにおける論理クロックの概念とその重要性を説明せよ。コンピュータシステムのクロックは決して逆戻りしてはならない。このことが論理クロック概念の実装においてどのように扱われているかを説明せよ。

6.12 本章で述べた相互排他のための集中型の方法において、コーディネータは待ち行列にいる最初のプロセスに対し臨界領域へ入ることを許可する。あるシステムでは、他の低優先度処理の前に高優先度処理に許可を与えることが望ましいかもしれない。これを考慮してアルゴリズムを修正し、そのアルゴリズムが非飢餓特性を満たしていることを示せ。

6.13 分散環境における相互排他実装のための最初の一般的アルゴリズムは Lamport[1978] によって開発された。このアルゴリズムの詳細を調べ、Ricart 及び Agrawala[1981] のアルゴリズムの性能及び信頼性と比較せよ。

6.14 相互排他のための過半数合意ベースの分散アルゴリズムは、臨界領域に入ろうとするプロセスが他のすべてのプロセスに要求メッセージを送信し、プロセスの過半数から許可を受信すると直ちに臨界領域に入る方式である。この

アルゴリズムの擬似コードを作成せよ。

6.15 デッドロックを説明せよ。デッドロックが発生するための4つの必要条件とは何か。もし4つの条件の1つが存在しない場合、デッドロックは起こり得ないことを示す適切な例を挙げよ。

6.16 一般的資源割り付けグラフにおけるサイクルの存在はデッドロックの存在の必要条件であるが、十分条件ではないことを証明せよ。

6.17 各資源のタイプに1つのユニットをもつシステムにおいて、資源割り付けグラフにおけるサイクルの存在はデッドロックの存在の必要十分条件であることを証明せよ。

6.18 与えられた資源割り付けグラフがデッドロックを含むか否か決定する擬似コードを書き表せ。

6.19 システムに3つの資源タイプ R_1, R_2, R_3 が存在し、それぞれに3、2、2ユニット存在する。4つのプロセス P_1, P_2, P_3、P_4 が並行してこれらの資源を次のように競合する。

(a) P_1 は R_1 の1つのユニットを保持し、R_2 の1つのユニットを要求している。

(b) P_2 は R_2 の2つのユニットを保持し、R_1 及び R_3 のそれぞれ1つのユニットを要求している。

(c) P_3 は R_1 の1つのユニットを保持し、R_2 の1つのユニットを要求している。

(d) P_4 は R_3 の2つのユニットを保持し、R_1 の1つのユニットを要求している。

このシステムの状態において、どのプロセスがデッドロック (存在するなら) になるか。

6.20 次のIPC（プロセス間通信）プリミティブを使用する分散システムがある。

(a) send(receiver_process_id, sender_process_id, message)

(b) receive(sender_process_id, message)

受信者がメッセージを受信する準備ができていないならば送信者はブロックする、及び送信者からメッセージが受信されるまで受信者はブロックするという意味でこれらのプリミティブは同期している。このシステムにおいて通信デッドロックが発生するための通信プロセスの最少数はどれだけか。回答数の理由を述べよ。最少数のプロセスを含む通信デッドロックの例を挙げよ。

6.21 演習問題6.20の分散システムにおいて、IPCが2つのタイプのプロセ

ス-クライアント及びサーバの間でのみ行われると仮定する。すなわち、一方が常にクライアントで、他方がサーバである。クライアントは常に (送信 (send), 受信 (receive)) のシーケンスで IPC プリミティブを実行する。すなわち、すべての IPC に対して、クライアントは最初にサーバへの要求メッセージを送信するために send を実行する。それからその要求に対する応答を受信するために receive を実行する。他方、サーバは常に (receive, send) のシーケンスで IPC プリミティブを実行する。すなわち、すべての IPC に対して、サーバは最初にクライアントからの要求を受信するために receive を実行する。それから処理の後に、クライアントに処理の結果を送信するために send を実行する。このシステムでデッドロックは起きうるか。その答えと理由を述べよ。

6.22 安全、不安全、及びデッドロックを区別せよ。システムにおいて、4 つのプロセス P_1, P_2, P_3、及び P_4 が競合する 4 ユニットの資源があるものとする。P_1, P_2, P_3、及び P_4 によって要求される資源の最大ユニット数はそれぞれ 3、6、5、及び 4 とし、現在それぞれ 2、1、3、及び 2 ユニットの資源を保持しているものとする。現在のシステムの状態は安全か不安全のどちらかであるかを見つけよ。もし安全であるならば、すべての安全シーケンスを挙げよ。

6.23 システムに 4 つのユニットの資源がある。3 つのプロセスがこの資源を競合し、各プロセスが最大 2 ユニット必要とする。これらのプロセスがどのような順序で、資源ユニットを獲得、解放しても、デッドロックが発生しないことを証明せよ。

6.24 不安全状態はデッドロック状態ではないことを証明せよ。

6.25 もし競合するプロセスに対して、資源が適切にスケジュールされなければ、システムは安全状態から不安全状態になるかもしれないことを示す例を挙げよ。次に、同じ例を用いて次の事を示せ。

(a) システムは不安全状態からデッドロック状態になるかもしれない。

(b) すべての競合するプロセスは不安全状態からデッドロック状態になることなく、正常に完了するかもしれない。

6.26 デッドロックを回避するためには、プロセスの資源使用に関する知識が不可欠である理由を論ぜよ。デッドロックの扱いにおいて、デッドロック回避方策が分散システムにおいて使用されない理由は何か。

6.27 デッドロックの必要条件の少なくとも 1 つが成り立たないように、システムを注意深く設計することによりデッドロックは予防されるかもしれない。この考えに基づき、次のそれぞれにつきデッドロック予防方式を示せ。

(a) 相互排他条件を否定する

(b) 保持待ち条件を否定する

(c) 循環待ち条件を否定する

(d) 先取り権なし条件を否定する

各方式の実際への適用可能性を論ぜよ。

6.28 次の資源割り付け方策はデッドロックを予防することを証明せよ。

(a) 順序要求

(b) 一括要求

6.29 先取り権付きの方法をデッドロックの予防に使用するシステムもある。システムが現在5つのトランザクション t_1，t_2，t_3，t_4，及び t_5 をもっており、それぞれのタイムスタンプは t_1，t_2，t_3，t_4，及び t_5 である ($t_1 > t_2 > t_3 > t_4 > t_5$)。次の場合にどうなるか。

(a) システムは待機消滅方式を使用し、t_2 は t_5 によって保持されている資源を要求する。

(b) システムは待機消滅方式を使用し、t_4 は t_1 によって保持されている資源を要求する。

(c) システムは待機負傷方式を使用し、t_3 は t_4 によって保持されている資源を要求する。

(d) システムは待機負傷方式を使用し、t_5 は t_2 によって保持されている資源を要求する。

6.30 幽霊デッドロックとは何か。分散システムにおいて幽霊デッドロックが生じる理由は何か。集中型デッドロック検出方式において、要求時転送方法がローカルコーディネータから中央コーディネータへの情報転送のために使用されているとする。このアルゴリズムも幽霊デッドロックを検出する可能性があることの例を示せ。

6.31 偽デッドロックを検出しない集中型デッドロック検出アルゴリズムがStuart 他 [1984] によって開発された。このアルゴリズムがどのようにして偽デッドロックの検出を防止しているのかを調べよ。

6.32 分散システムにおける、探査ベースのデッドロック検出分散アルゴリズムのための擬似コードを書き表せ。WFG ベースの分散アルゴリズムに比べて、このアルゴリズムの主な利点は何か。

6.33 デッドロックの結果として、プロセスが強制終了させられるか又はロールバックさせられ、それから再開させられるとき、どのような問題が生じるか。これらの問題を暑かるための適切な方法を提示せよ。デッドロック状態からの回復のために適切な犠牲者の選択に関係して、これらの問題からどのような結

論が導き出せるか。

6.34 検出されたデッドロックからの回復のための犠牲者の選択に関わる主な問題は何か。適切な犠牲者選任アルゴリズムを提示せよ。そしてそのアルゴリズムは飢餓問題にどのように留意しているか。

6.35 分散システムにおいて選任アルゴリズムが通常必要である理由は何か。LAN ベースの分散システムはブロードキャスト機能を持っている。このシステムにおいて使用される単純な選任アルゴリズムを提示せよ。

6.36 選任の発動は実際には、現在のコーディネータプロセスが故障するときのみ必要である。しかし、これはブリーアルゴリズムの場合には異なる。このアルゴリズムでは選任は故障したプロセスが回復するときにも発動される。これは本当に必要であるのか。もし必要であるならば、その理由を説明せよ。もしそうでないならば、現在のコーディネータが故障するときのみ選任を発動するようにブリーアルゴリズムの修正を提案せよ。

6.37 本章に述べたリングベースの選任アルゴリズムにおいて、一方向のリングが使用された。リングが双方向であると想定する。選任アルゴリズムはより効率的となるであろうか。もしそうならなければ、その理由を述べよ。もしそうなるならば、そのようなアルゴリズムを提案し、システムに n 個のプロセスがあると仮定して、2 つのアルゴリズムにおけるコーディネータの選任に必要なメッセージの数を比較せよ。

6.38 本章に述べたリングベースの選任アルゴリズムにおいて、2 つ以上のプロセスがほとんど同時にコーディネータが故障したことを検出し、それぞれのプロセスがリング上を選任メッセージを循環させるかもしれない。このことは選任において問題を生じないが、ネットワーク帯域の浪費となる。ただ 1 つのメッセージがリングを完全に循環し、他のメッセージは検出され、直ちに消滅されるように、このアルゴリズムを修正せよ。

6.39 2 つ以上のプロセスがほとんど同時にコーディネータが故障したことを検出するとき、コーディネータを選任するブリーアルゴリズムでは何が起きるか。

参考文献

[Adelstein and Singhal 1995] Adelstein, F., and Singhal, M., "Real-Time Causal Message Ordering in Multimedia Systems," In: Proceedings of the 15th International Conference on Distributed Computing Systems,

IEEE, New York (May-June 1995).

[**Agarwal and Abbade 1991**] Agarwal, D., and El Abbadi, A., "An Efficient and Fault-Tolerant Solution of Distributed Mutual Exclusion," ACM Transactions on Computer Systems, Vol. 9, Association for Computing Machinery, New York, pp. 1-20 (1991).

[**Badal 1986**] Badal, D. Z., "The Distributed Deadlock Detection Algorithm," ACM Transactions on Computer Systems, Vol. 4, No. 4, Association for Computing Machinery, New York, pp.320-337(1986).

[**Barborak et al. 1993**] Barborak, M., Malek, M., and Dahbura, A., "The Consensusu Problem in Fault-Tolerant Computing," ACM Computing Surveys, Vol. 25, Association for Computing Machinery, New York, pp. 171-220 (1993).

[**Bernstein et al. 1987**] Bernstein, P. A., Hadzilacos, V., and Goodman, N., Concurrency and Recovery in Database Systems, Addison-Wesley, Reading, MA, pp. 289-307 (1987).

[**Bracha and Toueg 1984**] Bracha, G., and Toueg, S., "A Distributed Algorithm for Generalized Deadlock Detection." In: Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing, Association for Computing Machinery, New York, pp. 285-301 (1984).

[**Bracha and Toueg 1987**] Bracha, G., and Toueg, S., "Distributed Deadlock Detection," Distributed Computing, Vol. 2, pp. 127-138 (1987).

[**Bulgannawar and Vaidya 1995**] Bulgannawar, S., and Vaidya, N. H., "Distributed K-Mutual Exclusion," In: proceedings of the 15th International Conference on Distributed Computing Systems, IEEE, New York (May-June 1995).

[**Carvalho and Roucairol 1983**] Carvalho, o.s. F., and Roucairol, G., "on Mutual Exclusion in Computer Networks," Communications of the ACM, Vol. 26, No. 2, Association for Computing Machinery, New York, pp. 146-147 (1983).

[**Chandy and Lamport 1985**] Chandy, K. M., and Lamport, L., "Distributed Snapshots: Determining Global States of Distributed systems," ACM Transactions on Computer Systems, Vol. 3, No. 1, Association for Computing Machinery, New York, pp.63-75 (1985).

[**Chandy and Misra 1982**] Chandy, K. M., and Misra, J., "A Distributed Algorithm for Detecting Resource Deadlocks in Distributed Systems,"

In: Proceedings of the ACM Symposium on Principles of Distributed Computing, Association for Computing Machinery, New York, pp.157-164 (August 1982).

[Chandy et al. 1983] Chandy, K. M., Misra, J., and Haas, L. M., "Distributed Deadlock Detection," ACM Transactions on Computer Systems, Vol. 1, No. 2, Association for Computing Machinery, New York, pp.144-156 (1983).

[Choudhary et al. 1989] Choudhary, A. N., Kohler, W. H., Stankovic, J. A., and Towsley, D., "a Modified Priority Based Probe Algorithm for Distributed Deadlock Detection and Resolution," IEEE Transactions on Software Engineering, Vol. SE-15, No. 1, pp. 10-17 (1989).

[Cidon et al. 1987] Cidon, I., Jaffe, J. M., and Sidi, M., "Local Distributed Deadlock Detection by Cycle Detection and Clustering," IEEE Transactions on Software Engieering, Vol. SE-13, No. 1, pp. 3-14 (1987).

[Coffman et al. 1971] Coffman, Fr., E. G. Elphick, M. J., and Shoshani, A., "System Deadlocks," ACM Computing Surveys, Vol. 3, No. 2, Association for Computing Machinery, New York, pp. 67-78 (1971).

[Coulouris et al. 1994] Coulouris, G. F., Dollimore, J., and Kindberg, T., Distributed Systems Concepts and Design, 2nd ed., Addison-Wesley, Reading, MA (1994).

[Cristian 1989] Cristian, F., "Probabilistic Clock Synchronization," Distributed Computing, Vol. 3, pp. 146-158 (1989).

[Cristian 1991] Cristian, F., "Understanding Fault-Tolerant Distributed Systems," Communications of the ACM, Vol. 34, Association for Computing Machinery, New York, pp. 56-78 (1991).

[Cristian and Fetzer 1995] Cristian, F., and Fetzer, C., "Fault-Tolerant External Clock Synchronization," In: Proceedings of the 15th International Conference on Distributed Computing Systems, IEEE New York (May-June 1995).

[Drummond and Babaoglu 1993] Drummond, R., and Babaoglu, O., "Low-Cost Clock Synchronization," Distributed Computing, Vol. 6, pp.193-203 (1993).

[Dubois et al. 1988] Dubois, M., Scheurich, C., and Briggs, F. A., "Synchronization, Coherence, and Event Ordering in Multiprocessors," IEEE Computer, Vol. 21, pp. 9-21 (1988).

[Elmagarmid 1986] Elmagarmid, A. K., "A Survey of Distributed Deadlock Detection Algorithms," ACM SIGMOD, Vol. 15, No. 3, pp. 37-45 (1986).

[Fidge 1991] Fidge, C., "Logical Time in Distributed Computing Systems," IEEE Computer, Vol. 24, pp. 28-33 (1991).

[Fredrickson and Lynch 1987] Fredrickson, N., and Lynch, N., "Electing a Leader in a Synchronous Ring," Journal of the ACM, Vol. 34, pp. 98-115 (1987).

[Garcia-Molina 1982] Garcia-Molina, H., "Elections in a Distributed Computing system," IEEE Transactions on Computers, Vol. C-31, No.1, pp.48-59 (1982).

[Garg 1996] Garg, V. K., principles of Distributed Systems, Kluwer Academic, Norwell, MA (1996).

[Goscinski 1991] Goscinski, A., Distributed Operating Systems, The Logical Design, Addison-Wesley, Reading, MA (1991).

[Gray et al. 1981] Gray, J. N., Homan, P., Korth, H. F., and Obermarck, R. L., "A Straw Man Analysis of the Probability of Waiting and Deadlock in Database Systems," Technical Report RJ 3066. IBM Research Laboratory, San Jose, CA (1981).

[Gusella and Zatti 1989] Gusella, R., and Zatti, S., "The Accuracy of the Clock Synchronization Achieved by TEMPO in Berkeley UNIX 4.3BSD," IEEE Transactions on Software Engineering, Vol. SE-15, No. 7, pp. 847-853 (1989).

[Jefferson 1985] Jefferson, D. R., "Virtual Time," ACM Transactions on Programming Languages and Systems, Vol. 7, No.3, pp.404-425 (1985).

[Knapp 1987] Knapp, E., "Deadlock Detection in Distributed Databases," ACM Computing Surveys, Vol. 19, No. 4, pp. 303-328 (1987).

[Kopetz and Ochsenreiter 1987] Kopetz, H., and Ochsenreiter, W., "Clock Synchronization in Distributed Real-Time Systems," IEEE Transactions on Computers, Vol. C-36, pp. 933-940 (1987).

[Lamport 1978] Lamport, L., "Time, Clocks, and the Ordering of Events in a Distributed System," Communications of the ACM, Vol.21, No. 7, pp. 558-565 (1978).

[Lamport 1990] Lamport, L., "Concurrent Reading and Writing og Clocks," ACM Transactions on Computer Systems, Vol. 8, pp. 305-310 (1990).

[[**Lamport and Smith 1984**] Lamport, L., and Smith, P. M., "Byzantine Clock Synchronization," In: Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing, Association for Computing Machinery, New York, pp. 68-74 (1984).

[**Lamport and Smith 1985**] Lamport, L., and Smith, P. M., "Synchronizing Clocks in the Presence of Faults," Journal of the ACM, Vol, 32, No. 1, Association for Computing Machinery, New York, pp. 52-78 (1985).

[**Lee and Kim 1995**] Lee, S., and Kim, J. L., "An Efficient Distributed Deadlock Detection Algorithm," In: Proceedings of the 15th International Conference on Distributed Computing Systems, IEEE, New York (May-June 1995).

[**Liskov 1993**] Liskov, B., "Practical Uses of Synchronized Clocks in Distributged Systems," Distributed Computing, Vol. 6, pp. 211-219 (1993).

[**Lockhart 1994**] Lockhart, Jr., H. W., OSF DCE: Guide to Developing Distributed Applicvations, IEEE Computer Society Press, Los Alamitos, CA (1994).

[**Maekawa et al. 1987**] Maekawa, M., Oldehoeft, A. E., and Oldehoeft, R. R., Operating Systems: Advanced Concepts, Benjamin/Cummings, White Plains, NY (1987).

[**Mattern 1993**] Mattern, F., "Efficient Algorithms for Distributed Snapshots and Global Virtual Time Apporoximation," Journal of Parallel and Distributed Computing, Vol. 18, No. 4, pp. 423-434 (1993).

[**Mills 1991**] Mills, D. L., "Internet Time Synchronization: The Network Time Protocol," IEEE Transactions on Communications, Vol. 39, No.10, pp.1482-1493 (1991).

[**Misra and Chandy 1982**] Misra, J., and Chandy, K. M., "A Distributed Graph Algorithm: Knot Detection," ACM Transactions on Programming Languages and Systems, Vol. 4, No. 4, pp.678-686 (1982).

[**Mitchell and Merritt 1984**] Mitchell, D. P., and Merritt, M. J., "A distributed Algorithm for Deadlock Detection and Resolution," In: Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing, Association for Computing Machinery, New York, pp. 282-284 (1984).

[**Natrajan 1986**] Natrajan, N., "A Distributed Scheme for Detecting Communication Deadlocks," IEEE Transactions on Software Engineering, Vol. SE-12, No. 4, pp. 531-537 (1986).

[Pratt and Nguyen 1995] Pratt, G. A., and Nguyen, J., "Distributed Synchronous Clocking," IEEE Transactions on Parallel and Distributed Systems, Vol. 6, No. 3, pp. 314-328 (1995).

[Ramanathan et al. 1990a] Ramanathan, P., Kandlur, D. D¿, and Shin, K. G., "Hardware-Assisted Software Clock Synchronization for Homogeneous Distributed Systems," IEEE Transactions on Computers, Vol.C-39, No. 4, pp. 514-524 (1990).

[Ramanathan et al. 1990b] Ramanathan, P., Shin, K. G., and Butler, R. W., "Fault-Tolerant Clock Synchronization in Distributed Systems," IEEE Computer, Vol. 23, No. 10, pp. 33-42 (1990).

[Raynal 1991] Raynal, M., "A Simple Taxonomy for Distributed Mutual Exclusion Algorithms," ACM Operating Systems Review, Vol. 25, pp.47-50 (1991).

[Raynal 1992] Raynal, M., "About Logical Clocks for Distributed Systems," ACM Operating Systems Review, Vol. 26, No. 1, pp.41-48 (1992).

[Ricart and Agrawala 1981] Ricart, G., and Agrawala, A. K., "An Optimal Algorithm for Mutual Exclusion in Computer Networks," Communications of the ACM, Vol. 24, No. 1, pp. 9-17 (1981).

[Roesler and Burkhard 1989] Roesler, M., and Burkhard, W. A., "Resolution of Deadlocks in Object-Oriented Distributed Systems," IEEE Transactions on Computers, Vol. 38, No. 8, pp. 1212-1224 (1989).

[Rosenberry et al. 1992] Rosenberry, W., Kenney, D., and Fisher, G., OSF DISTRIBUTED COMPUTING EBVIRONMENT, Understanding DCE, O'Reilly & Associates, Sebastopol, CA (1992).

[Rosenkrantz et al. 1978] Rosenkarantz, D. J., Stearns, R. E., and Lewis, P. M., "System Level Concurrency Control for Distributed Database Systems," ACM Transactions on Database Systems, Vol. 3, No. 2, pp.178-198(1978).

[Sanders 1987] Sanders, B. A. "The Information Structure of Distributed Mutual Exclusin," ACM Transactions on Computer Systems, Vol. 5, pp.284-299(1987).

[Shin and Ramanathan 1987] Shin, K. G¿, and Ramanathan, P.,"Clock Synchronization of aLarge Multiprocesdsor System in the Presence of Malicious Faults," IEEE Transactions on Computers, Vol. C-36, No. 1, pp. 2-12 (1987).

[Silberschatz and Galvin 1994] Silberschatz, a., and Galvin, P. B., Operating Systms Concepts, 4th ed., Addison-Wesley, Reading, MA (1994).

[Singh and Kurose 1994] Singh, S., and Kurose, J., "Electing 'Good' Leaders," Journal of Parallel and Distributed Computing, Vol. 21, pp. 184-201 (1994).

[Singhal 1989] Singhal, M., "Deadlock detection in Distributed Systems," IEEE Computer, Vol. 22, No. 11, pp. 37-48 (1989).

[Singhal and Shivaratri 1994] Singhal, M., and Shivaratri, N. G., Advanced Concepts in Operating Systems, McGraw-Hill, New York (1994).

[Sinha and Natrajan 1985] Sinha, M. K., and Natrajan, N., "A Priority Based Disutriburted Deadlock Detection Algorithm," IEEE Transactions on Software Engineering, Vol. SE-11, No. 1, pp.67-80 (1985).

[Srikanth and Teueg 1987] Srikanth, T. K., and Teueg, S., "Optimal Clock Synchronization," Journal of ACM. Vol. 34, No.3, pp. 626-645 (1987).

[Stalling 1995] Stalling, W., Qperating Systems, 2nd ed., Prentice-Hall, Englewood Cliffs, NJ (1995).

[Steinman et al. 1995] Steinman, J. S., Lee, C. A., Wilson, L. F., and Nicol, D. M., "Global Virtual Time and Distributed Synchronization," In: Proceedings of the 9th Workshop on Parallel and Distributed Simulation, pp. 139-148 (1995).

[Stuart et al. 1984] Stuart, D., Buckley, G., and Silberschatz, A., "A Centralized Deadlock Detection Algorithm," Technical Report, University of Texas at Austion (1984).

[Suzuki and Kasami 1985] Suzuki, I., and Kasami, T., "A Distributed Mutual Exclusion Algorithm," ACM Transactions on Computer Systems, Vol. 3, No. 4, pp.344-4349 (1985).

[Tanenbaum 1992] Tanenbaum, A. S., Modern Operating Systems, Prentice-Hall, EnglewoodCliffs, NJ (1992).

[Tanenbaum 1995] Tanenbaum, A. S., Distributed Operating Systems, Prentice-Hall, Englewood Cliffs, NJ (1995).

[Tel 1994] Tel, G., Introduction to Distributed Algorithms, Cambridge University Press, New York, NY (1994).

[Turek and Shasha 1992] Turek, J., and Shasha, D., "The Many Faces of Consensus in Distributed Systems," IEEE Computer, Vol. 25, pp. 8-17 (1992).

[Vasanthavada and Marinos 1988] Vasanthavada, N., and Varinos, P. N¿, "Synchronization of Fault-Tolerant Clocks in the Presence of Malicious Failures," IEEE Transactions on Computers, Vol. C-37, No. 4, pp.440-448(1988).

[Wuu and Bernstein 1985] Wuu, G. T., and Bernstein, A. J., "False Deadlock Detectio in Distributed Systems," IEEE Transactions on Software Engineering, Vol. SE-11, No. 8, pp.820-821 (1985).

[Yang and Marsland 1994] Yang, Z., and Marsland, T. A. (Eds.), Global States and Time in Distributed Systems, IEEE Computer Society Press, Los Alamitos, CA (1994).

インターネット参考文献

分散システムにおける時間に関する参考文献は、以下のところで見つけることができる。

ftp:ftp.cs.umanitoba.ca/pub/bibliographies/Distributed/dist-time.html

並行プロセスの同期に関する参考文献は、以下のところで見つけることができる。

ftp:ftp.cs.umanitoba.ca/pub/bibliographies/Parallel/par.synch.html

本章に関係する他のトピックスに関する参考文献は、以下のところで見つけることができる。

ftp:ftp.cs.umanitoba.ca/pub/bibliographies/Os/IMMD_IV.html

ftp:ftp.cs.umanitoba.ca/pub/bibliographies//Misc/misc.1.html

第7章

資源管理

7.1 はじめに

分散システムの特徴としては、**資源の多重性 (resource multiplicity)** 及び**システムの透過性 (system transparency)** がある。各々の分散システムはネットワークによって相互接続された多くの資源からなっている。通信機能を提供するのに加え、ネットワークは、ローカルプロセスを移動させ、ネットワークの遠隔ノードでそのプロセスを実行させることにより、資源共有 (resource sharing) を可能とする。ローカルノードが要求された資源を持っていない、又はローカルノードをシャットダウンしなければならないといった理由で、あるプロセスは移動させられるかもしれない。また、プロセスは期待されるターンアラウンド時間が改善されるならば、遠隔で実行されるかもしれない。ユーザの観点から見ると、分散システムにおける使用可能な資源の集合は、1つの仮想システムのように動作する。したがって、ユーザがあるプロセスの実行を要求すると、プロセスへ資源の割り当てを制御すること、及びこれらの割り当てに従ってシステムの適切なノードにプロセスを転送することは、分散オペレーティングシステムの資源管理者 (resource manager) の責任となる。資源は共有ファイルのように論理的であったり、CPU のように物理的であったりする。本書の目的のために、資源をシステムのプロセッサと考え、各プロセッサは分散システムのノードを形成すると仮定する。このようにして、本章では、ノード (node) とプロセッサ (processor) という用語を同じものを意味するものとして使用する。

資源管理者は、分散システムにおけるプロセスをスケジュールして、システ

ム資源の使用、応答時間、ネットワーク輻輳、スケジュールオーバヘッドを最適化する。分散システムのプロセスをスケジュールする様々な異なる技法及び方法が提案されてきた。これらの技法は大きく3つのタイプに分類できる。

1. **タスク割り付け方法 (task assignment approac)**:この方法では、ユーザから処理を要求された各プロセスは関連するタスクの集まりとして見られる。これらのタスクは性能を改善するように適切なノードに割り付けられる。
2. **負荷バランス方法 (load-balancing approach)**:この方法では、ユーザから要求されたすべてのプロセスは、ノード間の作業負荷を平準化するように、ノード間に分散される。
3. **負荷分担方法 (load-sharing)**:この方法は、処理されることを待っているアイドル状態にどのプロセスもないことを保証することによりシステムの処理能力を保存しようと試みる。

3つの方法のうちで、タスク割り付け方法は実際の状況では適用に限度がある。その理由は、スケジュールされるすべてのプロセスの特性はあらかじめ既知であるという仮定の上で動作するからである。さらに、この種類のスケジュールアルゴリズムは通常、システムの動的な状態変化に対応していない。それゆえ、この方法は動作原理を簡単に説明するにとどめる。これらの各技法を述べる前に、良好な全体スケジューリングアルゴリズムの望ましい特性について説明する。

7.2 全体スケジューリングアルゴリズムの望ましい特性

7.2.1 プロセスに関する事前知識が不要

良好なプロセススケジューリングアルゴリズムは実行されるプロセスについての事前の知識を絶対的に必要としないで運用すべきである。プロセスの特性及びプロセスが必要とする資源についての情報に基づき運用するスケジューリングアルゴリズムは通常、プロセスの実行を要求するときにこの情報を規定しなければならず、ユーザに余分の負荷を与えることとなる。

7.2.2　動的特性

良好なプロセススケジューリングアルゴリズムは、システムの様々なノードの動的な負荷変化 (又は状態変化) に対応できるべきである。すなわち、プロセス割り付けの決定は、固定の静的な方針に基づくのではなく、システムの現在の負荷に基づいてなされるべきである。このために、時にはスケジューリングアルゴリズムはあるプロセスを 1 回以上移動できることが推奨される。あるプロセスを特定のノードに割り付けるという最初の決定は、時間の経過とともに新しいシステム負荷に適応するため、変更される必要があるかもしれないからである。この特性は、システムが先取り権付きの**プロセス移動機能 (process migration facility)** をサポートすることを必要とする。この機能では、プロセスを 1 つのノードから別のノードにその実行途中に移動させることができる。

7.2.3　迅速な決定能力

良好なプロセススケジューリングアルゴリズムは、プロセスのプロセッサへの割り付けについて迅速な決定を下すべきである。これはアルゴリズムの極めて重要な側面であり、多くのアルゴリズム案はこのゆえに不適となっている。例えば、システムを数学的プログラムでモデル化し、それをオンラインで解くアルゴリズムはこの要件に合致しないので、不適である。発見的方法は少ない計算量で、準最適の結果を得ることができるので、厳密 (最適) な解法よりも好まれる。

7.2.4　システム性能とスケジューリングオーバヘッドとのバランス

幾つかの全体スケジューリングアルゴリズムは全体の状態情報を収集し、プロセス割り付けの決定にその情報を使用する。一般的な直感では、全体システム状態を記述するより多くの量の情報は、全体としてシステムにより良い効果をもたらすためのより知的なプロセス割り付けを可能とする。しかし、分散環境において、システムの状態に関する情報を収集するコストは集中型システムに比べて高価である。一般的な観察によると、システムの全体状態に関する情報をより多く得ようとするとオーバヘッドが増加するので、情報の収集コストと処理コストの結果として収集の頻度を制限せざるを得ず、そのため収集された情報が古くなり、スケジューリングの頻度も低下するので、情報の有効性は

減少する。したがって、全体情報を収集するオーバヘッドが最小で、準最適のシステム性能を提供するアルゴリズムが望ましい。

7.2.5 安定性

ここでは**安定性 (stability)** についてのべる。システムのすべてのノードがより良好な性能を求めプロセスをスケジュールしようとしているが、何等有効な処理結果を得られずにプロセスの移動に時間のすべてを費やしている状態に入ることがある場合には、スケジューリングアルゴリズムは、不安定 (unstable) であると言われる。この無駄なプロセスの移動は**プロセッサスラッシング (processor thrashing)** として知られている。プロセッサスラッシングは次の状況で起きうる。システムの各ノードが自己のプロセスをスケジュールする権限があり、スケジュールの決定が、他のプロセッサによって行われた決定と独立になされるか、又はノード間の転送遅れによる比較的古いデータに基づいてなされる場合。例えば、ノード n_1 と n_2 の両方がノード n_3 が空いていると観察し、他方によって負荷分散が決定されたことを気付かずに、それぞれの仕事の一部を n_3 に負荷分散しようとする場合に起きうる。もしノード n_3 がノード n_1 及び n_2 から渡されたプロセスによって過負荷になると、n_3 は再び他のノードにプロセスを渡そうとする。この全体のサイクルはいつまでも繰り返されるであろう、その結果不安定状態になる。これは良好なスケジューリングアルゴリズムにとって、望ましいことではない。

プロセッサスラッシングは、軽負荷ノードへ転送中のプロセスを考慮していない場合にも発生する。この場合、幾つかのプロセスが同じノードに移動させられるかもしれない、そして不安定の要因となる。例えば、特定の瞬間に、ノード n_1 が非常に忙しく、ノード n_2 が空いているとする。また、ノード n_1 は 2 秒ごとに起動され、プロセスがノード n_1 からノード n_2 に到達するのに平均 20 秒を要するものとする。その場合、最初のプロセスが受信される前にノード n_1 は少なくとも 10 個のプロセスをノード n_2 に送信することが考えられる。これは結果として不安定状態になりうる。この問題に対する 1 つの単純な方法はどのプロセスが最近送信されたかを追跡し、この情報を問題の軽減に、及びプロセス移動の最小化に使用することである。安定性を扱う、より高度な技法は可能であるが、さらに過去のデータを保持する必要があり、したがって高価なものとなる。

7.2.6　規模拡張性

スケジューリングアルゴリズムは小規模ネットワークでも大規模ネットワークでも同様に扱うことができるべきである。最初にすべてのノードから負荷状況を問い合わせ、それから最も軽負荷のノードをプロセスを受け入れる候補として選択するようなスケジュール決定アルゴリズムは**規模拡張性 (scalability)** のに問題がある。このようなアルゴリズムは小規模ネットワークでは良好に働くが、大規模ネットワークに適用しようとすると行き詰まる。これは問い合わせ元が、ほとんど同時に非常に多数の応答を受信し、その応答を処理しホストを選択するのに要する時間は、通常長すぎるからである。さらに、アルゴリズムの N^2(N はシステムのノードの総数) 特性は重いネットワークトラフィックを発生し、ネットワークバンド幅を急速に消費する。アルゴリズムをの規模拡張性を高める単純な方法はホストを選択するのに、N ノードのうち m 個のノードのみ探索することである。m の値は N の値に依存して動的に調整される。

7.2.7　フォールトトレラント性

良好なスケジューリングアルゴリズムはシステムの 1 つ以上のノードの故障によって使用不能になってはならない。どのような時でも、その時点で動作しているノードに対し機能を継続しなければならない。さらに、もしリンクの故障により、ノードが 2 以上のグループに分割されるなら、アルゴリズムはグループ内のノードに対して機能を継続できなければならない。分散型の決定機能をもち、決定を行う過程で使用可能なノードのみを考慮するアルゴリズムはより良い**フォールトトレラント性**の能力をもつ。

7.2.8　サービスの公正性

提供されるサービスの平均的品質は明らかに重要な性能指標であるが、サービスがどのように公平に割り付けられるかもまた一般的関心事である。それが**サービスの公正性**である。例えば、2 人のユーザが同じプロセスを同時に起動し、ほぼ同じサービスの品質を期待する。何人かの研究者は、システムのすべてのノードの平均負荷をむやみにバランスさせようとする全体スケジューリング方策は、サービスの公平性の観点からよくないと考える。これは、どのような負荷バランス方式でも、軽負荷ノードがスタンドアローンに比べ悪い応答時間となり、重負荷のノードはその利益を得るであろう。望ましいのは、軽負荷

ノードに過度に影響を与えることなく、重負荷ノードの応答時間を改善することである。このために、負荷バランスの概念を負荷分散 (load sharing) の概念に置き換えるべきであろう。すなわち、あるノードはそのユーザに重大な影響を与えない範囲で、その資源の幾つかを分散するという考えである。

7.3 タスク割り付け方法

7.3.1 基本的考え

この方法では、1つのプロセスは複数の**タスク (task)** から構成され、目標は個別のプロセスのタスクに対し、最適の割り付け方策を見出すことである。タスク割り付け処理において典型的な仮定を次に述べる。

■ 1つのプロセスはタスクと呼ばれる断片に既に分割されている。この断片は、それ自体で完結しタスク間で転送されるデータが最小になるように、自然な境界で分割されている。

■ 各タスクに要求される計算量及び各プロセッサの計算速度が既知である。

■ システムの各ノード上の各タスクを処理するコストが既知である。このコストは通常、各プロセッサの速度と各タスクに要する計算量に基づき導出される。

■ 各タスク対ごとの**プロセス間通信 (interprocess communication : IPC)** コストが既知である。IPC コストは同じノードに割り付けられたタスクの間ではゼロ (無視できる) とみなされる。これらのコストはプロセスの静的プログラムの分析によって通常推定される。例えば、プロセスの実行中に、もし2つのタスクが n 回通信を行い、各タスク間通信の平均時間が t であるならば、2つのタスクのタスク間通信コストは n × t である。

■ タスクの資源要求、各ノードにおける使用可能資源、タスク間の優先関係、その他の制約条件が既知である。

■ タスクの再割り付けは一般的に不可能である。

これらの仮定で、タスク割り付けアルゴリズムは次の目標を達成するような方法で分散システムのノードにプロセスのタスクを割り付けようとする。

■ IPC コストの最小化

- 完了したプロセスに対し、迅速なターンアラウンド (turnaround) 時間
- 高度な並列処理 (parallelism)
- 一般的に効率的なシステム資源の使用

これらの目標はしばしば互いに矛盾する。例えば、IPC の最小化は、プロセスのすべてのタスクを 1 つのノードに割り付けることになりがちであるが、システム資源の効率的な使用はタスクをノード間に平等に分散させようとする。同様に、高速のターンアラウンド時間と高度な並列処理はタスクの並行処理を促進するが、タスク間の優先関係は並行処理を制限する。また、m 個のタスクで、q 個のノードの場合にタスクをノードに割り付けるには、m^q 通りの組み合わせがある。しかし実際には、タスクをノードに割り付け可能な実際の数は、あるタスクはあるノードに割り付けできないという制限のため、m^q よりも少ないかもしれない。タスクには特定の資源要求があり、要求された資源はシステムのすべてのノードで使用可能とは限らないからである。

例を用いて説明するため、図 7.1 の割り付け問題を考える。これは Stone[1977] によって議論された問題と同じである。これは 2 つのみのタスク割り付けパラメータをもっている。タスク実行コストとタスク間通信コストである。このシステムは 6 個のタスク $t_1, t_2, t_3, t_4, t_5, t_6$ と 2 つのノード n_1, n_2 からなっている。タスク間コスト (c_{ij}) 及びタスクの実行コスト (x_{ab}) が、それぞれ図 7.1(a) 及び (b) の表形式で与えられる。図 7.1(b) の特定のノードに対して特定のタスクの無限大のコストは、そのタスクはそのノードで使用不可能な特定の資源を要求しているため、そのノードで実行できないことを意味する。このようにして、タスク t_2 はノード n_2 で実行できないし、タスク t_6 はノード n_1 で実行できない。分散計算システムのこのモデルでは、プログラム内のタスク実行における並列処理やマルチタスキング (multitasking) はない。したがって、プロセス実行の全体コストは、割り付けられたノードでのタスクの全体実行コスト及び異なるノードに割り当てられたタスク間通信コストからなっている。

図 7.1(c) は 2 つのノードへの逐次的な割り付けを示す。最初の 3 つのタスクはノード n_1 に割り付けられ、残りの 3 つのタスクはノード n_2 に割り付けられている。この割り付けは、全体の実行コストを最小にすることが目標である。しかし、もし実行コストと通信コストの両方が考慮されると、この割り付けに対するコストは 58 となる。図 7.1(d) は実行コストと通信コストの合計を最小にする 2 つのノードへのタスクの最適割り付けを示す。この場合、実行コストは前の割り付けの実行コストよりも大きいが、合計の割り付けコストは 38 の

タスク間通信コスト						
	t_1	t_2	t_3	t_4	t_5	t_6
t_1	0	6	4	0	0	12
t_2	6	0	8	12	3	0
t_3	4	8	0	0	11	0
t_4	0	12	0	0	5	0
t_5	0	3	11	5	0	0
t_6	12	0	0	0	0	0

(a)タスク間通信コスト

実行コスト		
タスク	ノード	
	n_1	n_2
t_1	5	10
t_2	2	∞
t_3	4	4
t_4	6	3
t_5	5	2
t_6	∞	4

(b) 2 つのノード上のタスクの実行コスト

逐次的割り付け	
タスク	ノード
t_1	n_1
t_2	n_2
t_3	n_3
t_4	n_4
t_5	n_5
t_6	n_6

(c)逐次的割り付け

最適割り付け	
タスク	ノード
t_1	n_1
t_2	n_2
t_3	n_3
t_4	n_4
t_5	n_5
t_6	n_6

(d)最適割り付け

逐次的割り付けの実行コスト(x) $= x_{11} + x_{21} + x_{31} + x_{42} + x_{52} + x_{62} = 5+2+4+3+2+4 = 20$

逐次適割り付けの通信コスト(c) $= c_{14} + c_{15} + c_{16} + c_{24} + c_{25} + c_{26} + c_{34} + c_{35} + c_{36}$

$= 0+0+12+12+3+0+0+11+0 = 38$

逐次適割り付けの合計コスト $= x+c = 20 + 38 = 58$

最適割り付けの実行コスト(x) $= x_{11} + x_{21} + x_{31} + x_{41} + x_{51} + x_{62} = 5+2+4+6+5+4 = 26$

最適割り付けの通信コスト(c) $= c_{16} + c_{26} + c_{36} + c_{46} + c_{56}$

$= 12+0+0+0+0 = 12$

最適割り付けの合計コスト$= x+c = 26 + 12 = 38$

図 7.1　タスク割り付け問題の例

みである。

7.3.2　最適割り付けの検出

実行コストと通信コストの総計を最小化するようなタスクのノードへの割り付け方を検出する問題は、 Stone [1977, 1978] 及び多くの研究者 [Lo 1988, Wu and Liu 1980, Bokhari 1979] によってネットワークフローモデル及びネットワークフローアルゴリズムを用いてエレガントに分析されている。この方法では、最適割り付けは、図 7.2 に示す静的割り付けグラフを作成することによって検出できる。このグラフでは、ノード (グラフの)n_1 及び n_2 は分散システムの 2 つのノード (プロセッサ) を表し、ノード t_1 から t_6 はプロセスのタスクを表す。タスク間のエッジの重みはタスク間の通信コストを表す。タスクノードをノード n_1 に結ぶエッジの重みはそのタスクのノード n_2 での実行コストを表し、その逆も同様である。

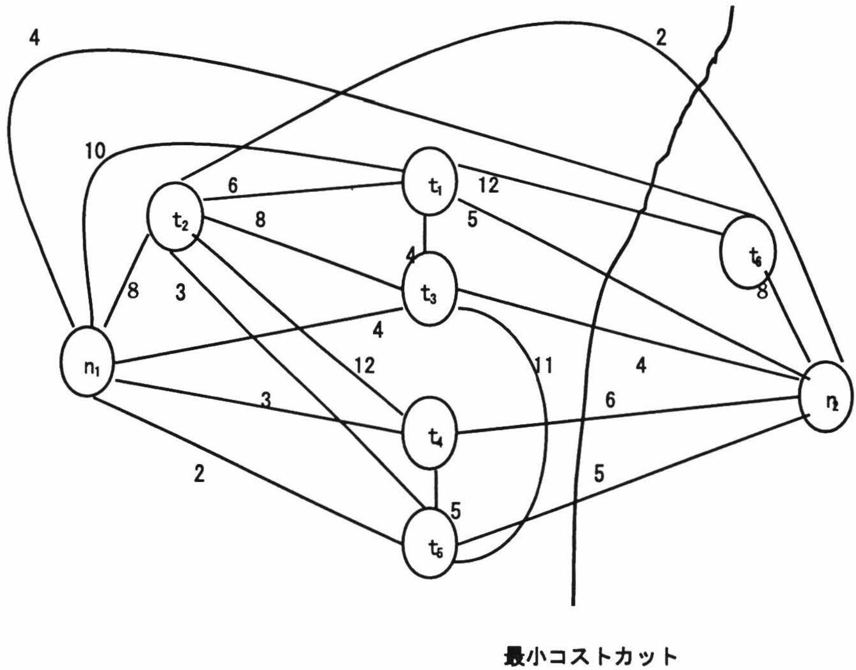


図 7.2　図 7.1 の割り付け問題を最小コストカットで割り付けたグラフ

このグラフの**カットセット (cutset)** は、これらのエッジが取り除かれたとき、グラフのノードが、一方の集合内のノードが n_1 から到達可能で、もう一方の集合内のノードは n_2 から到達可能である2つの分離した集合に分割されるようなエッジの集合であると定義される。各タスクノードは n_1 か n_2 のどちらからか到達可能である。カットセットのサブセットはカットセットにならない、すなわち、カットセットは最小集合である。各カットセットは1つのタスク割り付けに1対1で対応する。

カットセットの重みはカットセットに含まれるエッジの重みの合計である。カットセットの重みは、その割り付けに対する実行及び通信コストを合計するので、対応するタスク割り付けのコストを表す。最適の割り付けは最小の重みのカットセットを見つけることによって得られるであろう。これはネットワークフローアルゴリズムを使用することによりなされる。このアルゴリズムは比較的計算量の少ない部類に属する。図7.2における太線は、図7.1(d)の最適割り付けに対応する最小重みのカットセットを示す。もしタスク t_1 がノード n_2 に割り付けられているならば、ノード n_2 へのエッジは切断されるが、このエッジはタスク t_1 をノード n_1 で実行するコストに代わる。同様に、タスク t_1 とグラフの他のノードとの間の他のエッジは、タスク t_1 と他の対応するノードとの間の通信のためのこの割り付けによって起きる実際の通信コストを表す。

2プロセッサシステムでは、最適割り付けは多項式時間で**最大フロー／最小カット** (Max Flow/Min Cut) アルゴリズムを使用することにより見つけることができる [Shen 及び Tsai 1985]。しかし、任意の数のプロセッサに対して、**NP(Nondeterministic polynomial：非決定性多項式) 困難問題**であると知られている。NP 困難問題は、多項式時間で解くことができないので、計算が非常に困難である。したがって、より一般的な場合、何人かの研究者は、この問題を計算上の効率は良いが準最適解を求める発見的アルゴリズムに転換した。これらの発見的アルゴリズムに興味のある読者は [Aroa 及び Rana 1980, Efe 1982, Lo 1988] を参照されたい。

上記のモデルにおいて、1つのプロセスのタスクはシステムの様々なノードに割り付けられる。このモデルは、複数プロセスが割り付けられる一般的タスク割り付け問題に一般化されるであろう。この場合、各プロセスはプロセス群のタスクとして扱われ、プロセス間通信コストが既知であると仮定される。

上記の基本的タスク割り付けモデルの幾つかの拡張は文献において提案されている。基本的タスク割り付けモデルにおけるタスク実行コストとタスク間通信コストの2つのパラメータに加えて、拡張モデルでは、タスクに必要なメモ

リサイズ、プロセッサのメモリサイズ制約、タスク間の優先関係などの他のパラメータも考慮にいれている。しかし、実際の状況においてはタスク割り付け方法の適用に限界があるので、ここではこれ以上この話題を論じない。これらの拡張モデルに興味のある読者は [Lo 1988, Chu 及び Lan 1987, Rao 他 1979] を参照されたい。

7.4　負荷バランス法

この方法を用いるスケジューリングアルゴリズムは**負荷バランス (load-balancing) アルゴリズム**又は**負荷平準化 (load-leveling) アルゴリズム**として知られている。これらのアルゴリズムは、よりよい資源使用効率のためには、分散システムにおける負荷は平等にバランスしているのが望ましいという直感に基づいている。このようにして、負荷バランスアルゴリズムは、特定のシステム性能指標に対して全体としての性能を良くするために、負荷を重負荷ノードから軽負荷ノードに透過的に転送することにより、全体の負荷をバランスしようとする。ユーザの観点から性能を考えると、指標はプロセッサの**応答時間 (response time)** であることが多い。しかし、資源の観点から性能を考えると、指標は全体システムの**スループット (throughput)** である。応答時間と比較して、スループットでは、すべてのユーザは公平に扱われ、すべてが処理を進めているものとみなされる。すべての資源の使用効率を最大にするという資源の観点はシステムのスループットを最大にしたいということと同等である。このようにして、ほとんどの負荷バランスアルゴリズムの目標は全体システムのスループットを最大にすることである。

7.4.1　負荷バランスアルゴリズムの分類

ここでの分類は、負荷バランスアルゴリズムの特性の階層的分類である。分類の構造を図 7.3 に示す。特定の負荷バランスアルゴリズムを述べるために、分類はこの階層のパスをたどる。この分類について以下に説明する。

静的対動的

最高位のレベルで、負荷バランスアルゴリズムを静的 (static) と動的 (dynamic) に分類する。静的アルゴリズムはシステムの平均的特性についての情報のみ使用し、現在のシステムの状態については考慮しない。他方、動的ア

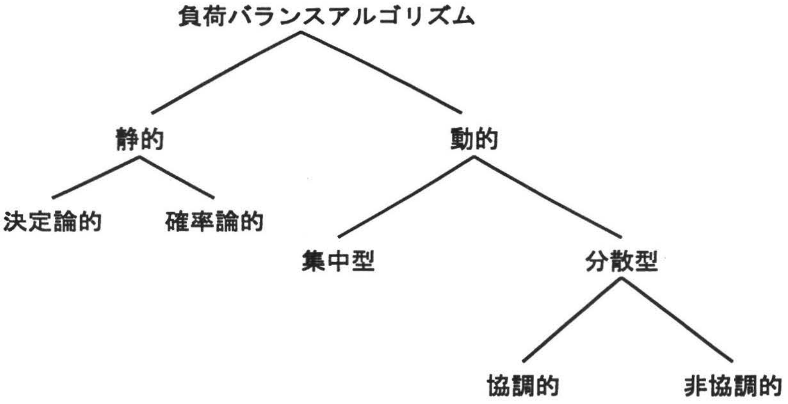


図 **7.3** 負荷バランスアルゴリズムの分類

ルゴリズムは動的に変化するシステム状態に反応する。

静的負荷バランスアルゴリズムは、システム状態の情報を維持し処理する必要がないので、単純である。しかし、現在のシステム状態に対応しないという事実によって、静的アルゴリズムの能力は制限される。動的アルゴリズムの魅力は、これらのアルゴリズムはシステム状態に反応し、それゆえ不必要に低い性能の状態になることを回避できることである。この理由のゆえに、動的な方策は静的な方策よりも極めて大きい性能の利点をもつ。しかし、動的アルゴリズムはシステム状態情報を収集し対応しなければならず、必然的に静的アルゴリズムよりも複雑となる。

決定論的対確率論的

静的負荷バランスアルゴリズムは決定論的 (deterministic) でも確率論的 (probabilistic) でもどちらもありうる。決定論的アルゴリズムは、決定論的にプロセスをノードに配置するために、ノードの特性及びプロセッサの特性についての情報を使用する。タスク割り付けアルゴリズムは基本的に決定論的静的負荷バランスアルゴリズムのカテゴリーに属する。

確率論的負荷バランスアルゴリズムは、ノード数、各ノードの処理能力、ネットワークトポロジー等のシステムの静的特性に関する情報を使用し、単純なプロセス配置ルールに定式化する。例えば、システムが 2 つのプロセス p_1 及び p_2 と 4 つの端末 t_1、t_2、t_3、及び t_4 をもつものとする。単純なプロセス

配置ルールとして、端末 t_1 及び t_2 から生じるすべてのプロセスはプロセッサ p_1 に、端末 t_3 及び t_4 から生じるプロセスはプロセッサ p_2 に割り付けることが考えられる。明らかに、このような静的負荷バランスアルゴリズムは、不必要に低い性能の状態になることを避ける能力に限界がある。例えば、端末 t_1 に特別のプロセスが発生するとき、プロセッサ p_1 が非常に重い負荷がかかっており、p_2 が空いているとする。このような状況ではプロセッサ p_2 がそのプロセスにとって、よりよい選択である。

一般的に、決定論的方法は最適化するのが困難で、実装にもコストがかかる。確率論的方法は実装が容易であるが、性能は良くないことが多い。

集中型対分散型

動的スケジューリングアルゴリズムは集中型 (centralized) 又は分散型 (distributed) である。集中型動的スケジューリングアルゴリズムにおいて、スケジューリングの責任は物理的に 1 つのノードに存在する。他方、分散型動的スケジューリングアルゴリズムでは、プロセス割り付けの決定を行う仕事は、システムの多くのノードの間に物理的に分散している。集中型の方法では、すべてのスケジューリングの決定がなされる 1 つのノードに、システム状態情報が集められる。このノードは集中型サーバノード (centralized server node) と呼ばれる。プロセススケジューリングのためのすべての要求は集中型サーバによって扱われる。サーバは格納した状態情報を用いて、新しいプロセスの配置について決定する。集中型の方法は、集中型サーバは各ノードにおける負荷及びサービスに必要なプロセスの数を知っているので、効率的なプロセス割り付けができる。基本的な方法では、他のノードは中央のサーバノードに、状態を更新するメッセージを周期的に送信する。これらのメッセージは集中型サーバノードにおけるシステム状態情報を最新のものに維持するために使用される。集中型サーバから他のノードに状態情報を問い合わせることも考えられる。これは、もしプロセスの割り付け要求のときに状態情報が使用されたなら、メッセージトラフィックを減少させる。しかしノードはローカルな活動によってその負荷を変えるので、古い状態情報の問題を引き起こすことになろう。

集中型機構の問題は信頼性の問題である。もし集中型サーバが故障すると、システムのすべてのスケジューリングが停止する。これに対する典型的な対策は、もし k 個の故障 (ノード故障) を生き残るには、$k+1$ 個のノードにサーバを複製することである。この方法において、すべての $k+1$ 個の複製サーバの一貫性を維持することはかなり高いオーバヘッドになるかもしれない。この

オーバヘッドを減少させる典型的な解決方法は厳密な一貫性を無しで済ませ、サーバの複数の複製を更新する単純な機構を使用することである。この方法は、厳密な一貫性がなくてもシステムが適切に機能する場合に適用できるという考えに基づく。

Theimer 及び Lantz [1989] は、もし数秒のサービスにおいて、時々起きる遅れが許容できるならば、集中型スケジューリング機構の信頼性を改善するために再インスタンシエーション (reinstantiation) という、より簡単な方法を使用できることを提案した。この方法では、$k+1$ 個のサーバの複製を維持する代わりに、1つのサーバが維持され、サーバ故障を検出するために監視を行う k 個のエンティティがある。故障が検出されると、サーバの新しいインスタンスが生成され、現在状態の更新を要求するマルチキャストメッセージを送信することにより、システムの状態情報を再構築する。スケジューリングサービスが停止している間の時間は、前のサーバの故障を検出する時間、新しいサーバにサーバプログラムをロードする時間、k 個のエンティティの間から複数で同時のインスタンシエーションを解決する時間、及び新しいサーバ上に全体の状態情報を再構築する時間の合計時間である [Theimer and Lantz 1989]。

集中型方式に比較して、分散型方式は1つのノードにスケジューリング処理をさせるという制約はない。1つのノードに状態情報を集めるという隘路を避け、システム状態の動的変化に迅速にスケジューラが反応することができる。分散型動的スケジューリングアルゴリズムは k 個の物理的な分散エンティティ $e_1, e_2, \ldots\ldots, e_k$ から構成される。各エンティティはローカルコントローラとみなされる。各ローカルコントローラは他とは非同期で並行に動作し、それぞれは、あらかじめ定まったノード集合のプロセスに対しスケジュールの決定に責任がある。各エンティティ e_i は、ローカルの目的関数に基づくのではなく、システム全体の目的関数に基づき決定を行う。各 e_i は他のエンティティと平等の基準で決定を行う。すなわち、短時間の間でもマスタエンティティは存在しない。完全分散アルゴリズムにおいて、各ノードは自身のエンティティをもつので、N 個のノードのシステムでは $k=N$ である。この場合、各エンティティは自己のノードのプロセスに対してスケジューリングの決定を行う責任をもつ。これらのプロセスにはローカルプロセスの転送と遠隔プロセスの受け入れ両者を含む。

協調的対非協調的

分散型動的スケジューリングアルゴリズムは協調的 (cooperative) と非協調

的 (noncooperative) に分類することもできる。非協調的アルゴリズムにおいて、個別のエンティティは自律的エンティティ(autonomous entity) として動作し、他のエンティティの動作とは独立にスケジューリングの決定を行う。他方、協調的アルゴリズムでは、分散エンティティはスケジューリングの決定を行うために互いに協調する。したがって、協調的アルゴリズムはより複雑であり、非協調的アルゴリズムよりもオーバヘッドが大きい。しかし、協調的アルゴリズムの安定性は非協調的アルゴリズムの安定性よりも良い。

7.4.2　負荷バランスアルゴリズムの設計問題

良好な負荷バランスアルゴリズムを設計することは以下の問題の理由で、困難な仕事である。

■ **負荷推定方策 (load estimation policy)**、システムの特定ノードの作業負荷をどのように推定するかを決定する。

■ **プロセス転送方策 (process transfer policy)**、プロセスをローカルに実行するか遠隔で実行するかを決定する。

■ **状態情報交換方策 (state information exchange policy)**、ノード間でシステムの負荷情報をどのようにして交換するかを決定する。

■ **配置方策 (location policy)**、転送することを決定したプロセスをどのノードに送るかを決定する。

■ **優先度割り付け方策 (priority assignment policy)**、特定ノードにおけるローカルプロセス及び遠隔プロセスの実行の優先度を決定する。

■ **移動制限方策 (migration limiting policy)**、プロセスが 1 つのノードから別のノードに移動する合計回数を決定する。

これらを以下に議論する。この議論のために、システム内のプロセスを 2 つの種類、すなわちローカルプロセスと遠隔プロセスに分類する。ローカルプロセスはそれが発生したノードで実行されるもので、遠隔プロセスは発生したノードとは別のノードで実行されるものである。外界からあるノードに到着した新しいプロセスは、そのノードによって実行することが認められると、ローカルプロセスとなる。認められなければ、そのプロセスはネットワークを通して転送され、宛先ノードにおける遠隔プロセスとなる。

負荷推定方策

負荷バランスアルゴリズムの主な目標はシステムのすべてのノード上での作業負荷をバランスさせることである。しかし、作業負荷をバランスさせようとする前に、特定のノードの作業負荷をどのようにして計測するかを決定する必要がある。したがって、どのような負荷バランスアルゴリズムでも最初の問題は、特定のノードの作業負荷を推定するための方法を決定することである。特定ノードの作業負荷の推定は、完全に満足できる解決方法が存在しないような困難な問題である。あるノードの作業負荷は幾つかの測定可能なパラメータに基づき推定される。これらのパラメータは次のような、時間依存やノード依存の要素を含んでいる。

- 負荷推定時点のノード上のプロセスの総数
- これらのプロセスの必要資源
- これらのプロセスの命令ミックス
- ノードのプロセッサのアーキテクチャと性能

負荷の計測はかなりの頻度で生じ、負荷はノードの現在状態を反映しているので、この計算は効率的でなくてはならない。これにより、すべてのパラメータがたとえ相対的な重要性がわかっていても、すべてを使用することはない。したがって、幾つかの負荷バランスアルゴリズムは、ノードの作業負荷の計測として、ノード上に現れているプロセスの総数を使用する。しかし、設計者によっては、これらのプロセスの残留サービス時間 (remaining service time) に依存して実際の負荷は大きく変わるので、このような推定は不適切な計測であると信じている。それゆえ、ノードの作業負荷を推定するために使用される別の計測は、そのノードの上のすべてのプロセスの残留サービス時間の合計とする方法である。しかし、この場合、解決すべき別の問題はプロセスの残留サービス時間をどのようにして推定するかである。Bryant 及び Finkel [1981] はこの目的のために次の方法の 1 つを使用することを提案した。

1. **メモリ無し法 (memoryless method)**:この方法は、すべてのプロセスが、今まで使用された時間に依存せずに同じ平均残留サービス時間をもつと仮定する。プロセスの残留サービス時間の推定のためのこの方法を使用することにより、基本的に負荷の推定はプロセスの総数の推定に帰することになる。
2. **過去同等法 (pastrepeat)**:この方法はあるプロセスの残留サービス時間は

それが今まで費やした時間に等しいと仮定する。

3. 確率分布法 (distribution method):もしサービス時間の確率分布が既知であるなら、あるプロセスの残留サービス時間は、今まで使用した時間を条件とした、平均残留サービス時間である。

プロセスの総数を計数する方法も、すべてのプロセスの残留サービス時間の合計をとる方法も、どちらも現代の分散システムにおける負荷推定方策として使用するのは適切ではない。その理由は、現代の分散システムにおいて、空きノードでさえも、メール及びニュースデーモンや、ウインドウマネジャなど幾つかのプロセスが常駐しているからである。さらには、これらのデーモンプロセスは周期的に動作し、行うべき仕事がないか検査する。なければ、再び休眠する。それゆえ、これらのシステムでローカルの推定方策として使用できる方法は、ノードの CPU の占有率を計測することである [Tanenbaum 1995]。**CPU 占有率** (central processing unit (CPU) utilization) は実時間の単位時間あたり実際に実行される CPU サイクルの数として定義される。明らかに、重負荷ノードの CPU 占有率は軽負荷ノードの CPU 占有率よりも大きい。あるノードの CPU 占有率は CPU 状態 (空き／使用中) を周期的に観察するためにタイマーを設定することによって計測される。

プロセス転送方策

負荷バランスアルゴリズムの方策は、重負荷のノードから軽負荷のノードに、あるプロセスの処理のためにプロセスを転送させるという考えに基づく。しかし、これを実現するために、ノードが軽負荷か重負荷かを決定するための方策を備える必要がある。ほとんどの負荷バランスアルゴリズムはこの決定のために閾値方策 (threshold policy) を使用している。ノードの閾値はその作業負荷の限界値であり、あるノードが軽負荷か重負荷かの判断に使用される。このようにして、あるノードにおいてその時点で作業負荷が閾値より低いならば、そのノードにおける新しいプロセスがローカルに受け入れられる。そうでなければ、軽負荷のノードにそのプロセスを転送しようとする。ノードの閾値は次の方法のいずれかによって決定されるであろう。

1. 静的方策 (static policy):この方法では、各ノードはその処理能力に応じてあらかじめ定まった閾値をもっている。この閾値はローカルノードでも遠隔ノードでも動的な作業負荷の変化に追従しない。この方法の主な利点は、閾値

を決定するためのノード間で状態情報の交換がないことである。

2. **動的方策 (dynamic policy)**:この方法では、ノード (n_i) の閾値は、すべてのノードの平均作業負荷とあらかじめ定まった定数 (c_i) との積として計算される。各ノード n_i に対して、値 c_i は、他のすべてのノードの処理能力に比較して、ノード n_i の処理能力に依存する。この方法において、ノードは状態情報交換方策 (後に述べる) の 1 つを使用して状態情報を交換する。動的方策は各ノードにより現実的な閾値を与えるが、状態情報を交換するためのオーバヘッドはその適用性を懐疑的にする。

ほとんどの負荷バランスアルゴリズムは単一の閾値を使用する、それゆえ、過負荷領域と軽負荷領域をもつのみである (図 7.4(a))。この単一の閾値方策では、もしその負荷が閾値よりも低いならば、あるノードは新しいプロセス (ローカル又は遠隔いずれでも) を受け入れる。また、もしその負荷が閾値を越えているならば、ローカルのプロセスを転送しようとするし、遠隔からの実行要求は廃棄する。それゆえ、ローカルプロセスを転送するという決定と遠隔プロセスを受け入れるという決定の両方は単一の閾値に基づいてなされる。単一の閾値の使用は無益なプロセス転送となるかもしれない、あるノードの負荷は、遠隔プロセスを受け入れると決定するときには、閾値よりも低いが、遠隔プロセスが到着すると直ちに閾値よりも大きくなるかもしれないので、スケジューリングアルゴリズムを不安定にする。それゆえ、遠隔プロセスを受信すると直ちに、ノードは 1 つ以上のプロセスを他のノードに転送しようとするであろう。さらに、 Alonso 及び Cova [1988] は次のように観察した。

■ あるノードは、転送によって残りのローカルプロセスの性能が大いに改善されるときに限り、1 つ以上のプロセスを転送すべきである。

■ あるノードは、遠隔プロセスの受け入れによる負荷がローカルプロセスの処理に著しく影響しないときに限り、遠隔プロセスを受け入れる。

単一の閾値の安定性を増すため、及び上記の 2 つの点に注意するため、Alonso 及び Cova [1988] は**高低方策 (high-low policy)** と呼ばれる 2 重閾値方策を提案した。図 7.4(b) に示されるように、高低方策は高マーク (high mark) と低マーク (low mark) と呼ばれる 2 つの閾値を使用する。高マークと低マークはノードの負荷状態を 3 つの領域に分割する。

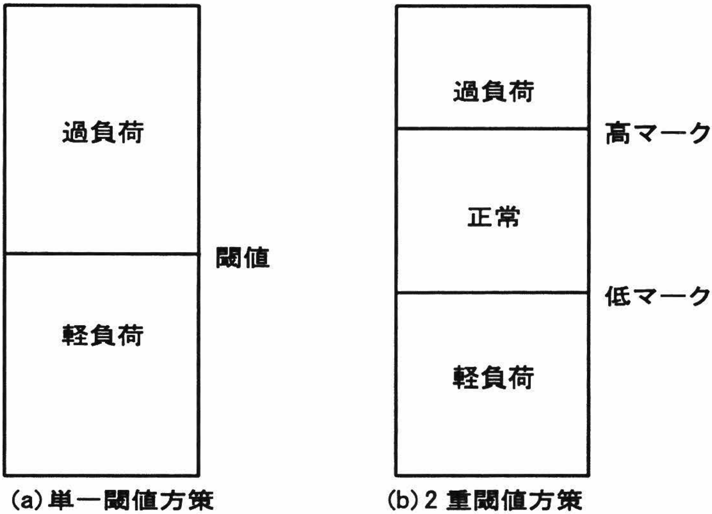


図 7.4　負荷領域

- ■ 過負荷 (overloaded): 高マーク値及び低マーク値を越える
- ■ 正常 (normal): 低マーク値を越え、高マーク値を下回る
- ■ 軽負荷 (underloaded): 両方のマーク値を下回る。

あるノードの負荷状態は図 7.5 に示されるように、1 つの領域から別の領域に動的に変化する。

あるノードの現在の負荷状態に依存して、ローカルプロセスを転送するという決定又は遠隔プロセスを受け入れるという決定は次の方策に基づいている [Alonso and Cova$_1$988]。

■ ノードの負荷が過負荷領域にあるとき、新しいローカルプロセスは遠隔で動作させるために転送させられる。また遠隔プロセスを受け入れる要求は棄却される。

■ ノードの負荷が正常領域にあるとき、新しいローカルプロセスはローカルで動作し、遠隔プロセスを受け入れる要求は棄却される。

■ ノードの負荷が軽負荷領域にあるとき、新しいローカルプロセスはローカ

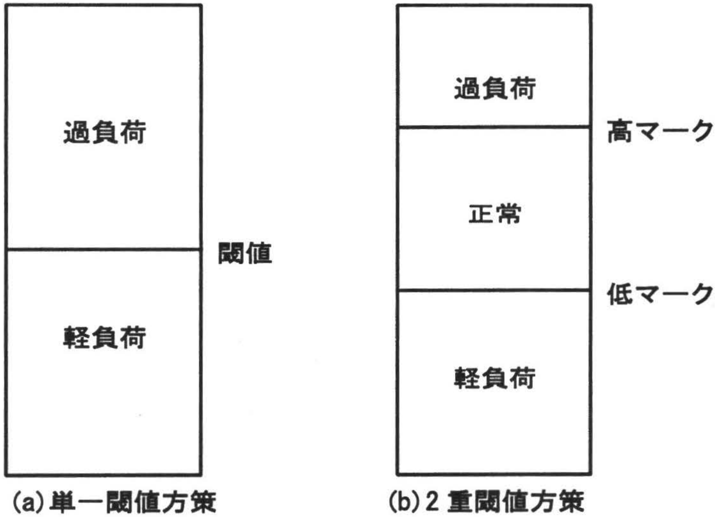


図 7.5 2 重閾値方策の場合にあるノードの負荷の状態遷移図

ルで動作し、遠隔プロセスを受け入れる要求は認められる。

高低方策はノード所有者にあらかじめ定めたレベルの性能を保証する。負荷バランスアルゴリズムは、遠隔プロセスの転送及び受信において発生するオーバヘッドを考慮する必要がある。あるプロセスは、転送の価値がない場合には他のノードに転送されないであろう。そして遠隔プロセスは、それを扱うのに余分の能力が十分にあるときに限って受け入れられる。

配置方策:あるノードからプロセスを転送するために転送方策によって決定がなされると、次のステップはそのプロセスの実行のための宛先ノードを選択することである。この選択はスケジューリングアルゴリズムの配置方策 (location policy) によってなされる。文献に提案された主な配置方策を以下に述べる。

閾値方策:この方法では、宛先ノードはランダムに選択され、そのノードへのプロセスの転送がそのノードにとって遠隔プロセスの受け入れが可能な状態なのかどうか決定するための検査がなされる。受け入れ可能ならば、そのプロセスは選択されたノードに転送される。そのノードはプロセスが実際に到着する

と、その状態に拘わらずそのプロセスを実行しなければならない。他方、選択されたノードが遠隔プロセスの受け入れを禁止するような状態にあると、別のノードがランダムに選択されて、同じように探査がなされる。これは、適切な宛先ノードが見つかるまで、又は探査の回数が、一定値の探査の上限 L_p を越えるまで行われる。後者の場合、プロセスを発生したノードがプロセスを実行しなければならない。Eager 他. [1986b] は、単一静的閾値転送方策及びこの閾値配置方策を用いてシミュレーションを行った。彼らのシミュレーション結果は、この方策の性能は探査上限値に余り依存しないことを示した。小さい (経済的でもある) 探査上限値 (例えば 3 又は 5) は大きい探査上限値 (例えば 20) の性能とほぼ同じである。シミュレーション結果はまた、閾値方策は負荷バランスを考慮しない方式に比較して本質的な性能改善を提供していることを示した。これは非常に単純な方式でも目立った利益をもたらすことができることを示している。

最小方策: この方法では、L_p 個の別々のノードが選択され、それぞれが負荷を決定するために順番にポーリングされる。プロセスはノードの負荷が遠隔プロセスを受け入れることができて、最小の負荷値 (minimum load value) をもつノードに転送される。ポーリングされたノードのどれもがプロセスを受け入れないならば、それを発生したノードで実行される。もし宛先ノードが見つかり、プロセスがそこに転送されると、プロセスが実際に到着する時点での状態に拘わらず、宛先ノードはそのプロセスを実行しなければならない。基本的な最小方策に対する単純な改善は、負荷がゼロのノードに遭遇すると探査を中断することである。この理由は、そのノードは受け入れ可能なことが保証されたノードであるからである。最小方策は、閾値方策よりも複雑な方法で、より多くの状態情報を使用する。しかし Eager 他. [1986b] は、彼らのシミュレーションを通して、最小方策の性能は、より単純な閾値方策の性能よりも著しく良くはないことを示した。これは、閾値方策に使用される以上の状態情報又は状態情報のより複雑な使用はほとんど効果がないことを示している。

入札方策: この方法では、システムは、サービスの買い手 (buyer) と売り手 (seller) が存在する**分散型計算経済 (distributed computational economy)** にたとえられる [Waldspurger 他 1992, Malone 他 1988]。ネットワークの各ノードは**入札プロセス (bidding process)** に関して 2 つの役割である管理者と契約者に責任をもっている。管理者は、あるノードが実行すべき場所を必要

とするプロセスをもっていることを示し、契約者は、遠隔プロセスを受け入れられるノードを示す。1つのノードはこれらの両方の役を引き受けるし、どのノードも厳密に管理者のみでもなく、契約者のみでもない。プロセスのためのノードを選択するために、管理者はシステムの他のすべてのノードに、**入札要請 (request-for-bids)** メッセージをブロードキャストする。このメッセージを受信すると、契約者ノードは管理者ノードに**札 (bid：入札の札)** を返す。札は入札価格を含んでおり、入札価格は契約者ノードの処理能力、メモリサイズ、資源稼働率、その他に基づき変わる。契約者ノードから受信した札のうちから、管理者ノードは**最善の札 (best bid)** を選ぶ。管理者の要求に対し最善の札は、要求がなされたアプリケーションについての最も安価、最も高速、又は最善の価格性能を意味する。最善の札が決定されると、プロセスは管理者ノードから**契約獲得者ノード (winning contractor node)** に転送される。しかし、契約者ノードは他の多数の管理者ノードからの多くの入札を同時に獲得するかもしれない、したがって過負荷になるかもしれない。この状況を防ぐために、最善の札が選択されると、その札の所有者にメッセージが送信される。その時点で、入札者はそのプロセスを受け入れ又は拒絶することが選択できる。メッセージが管理者ノードに返信されて、プロセスが受け入れられたか拒絶されたかを知らせる。契約者ノードは、入札が行われた時点と札が獲得されたと知らされる時点までの間で生じる状態の変化を理由に、獲得した札を棄却するかもしれない。もし札が棄却されると、入札手続きは最初から再びやり直される。

入札アルゴリズムは、全体のスケジューリング過程に参加するかどうかを完全にノードの自律性にまかされているので、興味深い。例えば、管理者ノードは契約者ノードにプロセスを送るかどうか決定する権利をもっている。契約者ノードは入札でそれに応え、遠隔プロセスを受け入れるかどうか決定する権利をもっている。契約者ノードは、そうすることを選ばなければ、遠隔プロセスの受け入れを強要されることはない。他方、入札アルゴリズムの2つの主な欠点は、通信オーバヘッドが大きいことと、及び適切な価格付け方策を決定することが困難であることである。両者の要素は入札の間に交換される情報の量とタイプの選別を必要とする。入札アルゴリズムに対して、交換される情報の量とタイプは一般に、アルゴリズムの有効性と性能のバランスをとるような方法で決定される。決定を行うために交換される情報の量とタイプに関しては様々である [Stankovic 及び Sidhu 1984, Smith 1980, Hwang 他 1982, Stankovic 1984]。

ペアリング方策: 今までに述べた方策に使われた負荷バランスを達成する方法は、システムのすべてのノードの負荷のバランスをとり偏差を減少することであった。これに対し、ペアリング方策により負荷バランスを達成する方法は、システムのノードの**ペア (pair：対)** の間でのみ負荷の偏差を減少させるものである。この配置方策は Bryant 及び Finkel [1981] によって提案された。この方法において、負荷が大きく異なる 2 つのノードが互いに一時的にペアにされ、同じペアに属するノード間で、負荷バランスをとる操作が実施される。この操作は 1 つ以上のプロセスを重負荷ノードから他方に移動することによってなされる。複数のノードペアがシステム内に同時に存在することもある。あるノードは少なくとも 2 つのプロセスをもっている場合に、相手を見つけようとする。そうでなければ、このノードからのプロセスの移動は意味がない。しかし、すべてのノードはペアリングの要求には好意的に応える。

基本的なペアリング方法では、各ノードはランダムに選別したノードにペアとなるかどうかを問い合わせる。回答を待つ間、質問者は他からの質問を拒絶する。もし意図した相手から応答を受けると、ペアが形成される。ペアが形成されると、これらの 2 つのノードの負荷をバランスするために、1 つ以上のプロセスがより重い負荷のノードから他方に移動させられる。移動させられるプロセスは相手のノードで処理完了までの期待時間 (平均時間) を比較することによって選択される。移動の遅れ時間はこの推定に含まれる。相手におけるサービス時間対自己におけるサービス時間の値が最大のプロセスが最初に送られるべきものとして選択される。他のプロセスを移動させるという決定は、最初のプロセスが相手のノードに受け取られ、相手のノードの負荷がこのプロセスが入ることによる影響が考慮されたという仮定に基づく。プロセスの移動が終わると直ちにペアは解消される。ペアが形成されている間、ペアの両メンバーは他からのペアリング (ペア形成) 要求を拒絶する。ペアリング機構の幾つかの変形は [Bryan and Finkel 1981] に見出すことができる。

状態情報交換方策

動的な方策はシステムのノード間で状態情報の頻繁な交換を必要とする。実際、ある動的負荷バランスアルゴリズムは、2 つの反対のインパクト、すなわちメッセージの転送はシステムの全体の性能に影響するので、転送のディレンマに直面する。一方では、転送は負荷をバランスさせるアルゴリズムの能力を改善する。他方では、通信チャネルの占有率が増加するので、メッセージの平均待ち時間が上昇する。このようにして、適切な状態情報交換方策の選択は重

要である。提案された負荷バランスアルゴリズムはこの目的のために次の方策の1つを使用する。

周期的ブロードキャスト法:この方法において各ノードは、時間の t 単位の経過ごとに自己の状態情報をブロードキャストする。明らかに、より大きいネットワークトラフィックを発生するという理由、及び無効なメッセージ(時間のt単位直前と状態が変わっていないノードからのメッセージ)がブロードキャストされる可能性があるという理由で、この方法は良くない。さらには、多くのノードをもつネットワークに対して状態情報交換のために生成されるメッセージの数は余りにも多いので、この方法の規模拡張性は良くない。

状態変化時ブロードキャスト法:この方法は周期的ブロードキャスト法の無効なメッセージ交換の問題を避ける。あるノードは、ノードの状態が変化する時のみ、その状態情報をブロードキャストする。外界から又はシステム内のどれか他のノードからプロセスが到着するとき、ノードの状態は変化する。プロセスは、実行が完了するとき、又はどれか他のノードに転送されるとき、そのノードを離れる。

この方法の改良は、あるノードの状態の些細な変化は他のすべてのノードに報告の必要がないことを観察することによって得られる。ノードは負荷バランスをとる過程において、軽負荷又は過負荷の場合にのみ関わるからである。それゆえ、改善された方法において、ノードの状態が正常負荷領域から軽負荷領域又は過負荷領域に変化するときのみ、その状態情報をブロードキャストする。明らかに、この改善された方法は2重閾値転送方策にのみ適用される。

要求時交換法:あるノードは、軽負荷又は過負荷になるときのみ他のノードの状態を知る必要がある。状態情報の要求時交換の方法はこの観察に基づいている。この方法において、あるノードは自己の状態が正常負荷領域から軽負荷領域又は過負荷領域に変化するとき、状態情報要求(StateInformationRequest)メッセージをブロードキャストする。このメッセージを受信すると、他のノードは要求したノードに現在の状態を送信する。この方法も2重閾値転送方策においてのみ動作する。

要求を行うノードはすべての他のノードの状態情報を必要としていないことを観察することにより、メッセージ数のさらなる削減がこの方法においても可能である。すべてというよりも、要求ノードにとって、負荷バランスに協力し

てくれるノードの状態情報のみが有用である。すなわち、要求を行うノードが軽負荷であるならば、過負荷ノードのみが負荷バランス過程に協力する。その逆も同様である。それゆえ、改善された要求時交換方策において、要求ノードの状態は状態情報要求メッセージに含まれている。状態情報要求メッセージを受信すると、負荷バランス過程において要求ノードに協力できるノードのみが応答する。それ以外のノードは応答を返さない。

ポーリングによる交換法: 上記の方法のすべてはブロードキャストを行う方法を使用している、そのため規模拡張性に乏しい。ポーリング機構は、ブロードキャストプロトコルを避けることにより、この制限を克服している。この方法は、あるノードがシステム内のすべての他のノードと状態情報を交換する必要はないという考えに基づいている。あるノードが負荷バランスのために他のどれかのノードの協力が必要なときは、他のノードを 1 つずつランダムなポーリングによって適切な相手を探すことができる。それゆえ、状態情報はポーリングを行ったノードと行われたノードとの間でのみ交換される。ポーリング過程は、適切な相手がみつかるとき、又はあらかじめ定まったポーリング回数の上限に達するとき終了する。

優先度割り当て方策:

プロセス移動が分散オペレーティングシステムによってサポートされているとき、特定のノードにおけるローカル及び遠隔プロセスの両者のスケジューリングのために、優先度割り当てルールを定める必要がある。以下の優先度割り当てルールのうちの 1 つがこの目的のために使用されるかもしれない。

1. 利己的 (selfish):ローカルプロセスが遠隔プロセスよりも高い優先度が与えられる。

2. 利他的 (altruistic):遠隔プロセスがローカルプロセスよりも高い優先度が与えられる。

3. 中間的 (intermediate):プロセスの優先度は、該当するノードにおけるローカルプロセス及び遠隔プロセスの数に依存する。もしローカルプロセスの数が遠隔プロセスの数以上であるならば、ローカルプロセスは遠隔プロセスよりも高い優先度が与えられる。そうでなければ、遠隔プロセスがローカルプロセスよりも高い優先度が与えられる。

Lee 及び Towsley [1986] はこれらの 3 つの優先度の割り当て方策の効果を全体の応答時間性能に関して研究した。彼らの研究の結果を以下に述べる。

■ 利己的優先度割り当てルールは 3 つの方策のうちで、最悪の応答時間性能となる。この方策では、遠隔プロセスの性能が極端に悪いからである。しかし、ローカルプロセスに対しては最善の応答時間性能となる。結果として、この方策は負荷の高いノードに到着するプロセスには厳しい罰則が課せられ、負荷の軽いノードに到着するプロセスは特典が与えられることになっている。

■ 利他的優先度割り当てルールは 3 つの方策のうちで最良の応答時間性能を達成する。しかし、この方策では、遠隔プロセスの方がローカルプロセスよりも遅れ時間が小さい。これは、ローカルプロセスは各ノードにおける第一義 (principal) の作業負荷であり、遠隔プロセスは第二義 (secondary) の作業負荷であるという意味では、幾らか非現実的である。

■ 中間的優先度割り当てルールの性能は上記の 2 つの中間に位置する。興味深いことに、この方策の全体の応答時間性能は利他的方策の性能にずっと近い。この方策では、システムの広い使用範囲において、遠隔プロセスよりもローカルプロセスが優遇される。

移動制限方策

プロセス移動 (process migration) をサポートする分散オペレーティングシステムによって使用されるもう 1 つの重要な方策は、移動が許容される回数の最大総数を定めることである。次の 2 つの方策のうちの 1 つがこの目的のために使用されるであろう。

無統制 (uncontrolled): この場合、あるノードに到着する遠隔プロセスは、そのノードで発生するプロセスと同様に扱われる。それゆえ、この方策では、プロセスは何回でも移動される。この方策は不安定性を生じる欠点をもっている。

統制 (controlled): 無統制方策の不安定問題を克服するために、ほとんどのシステムは遠隔プロセスをローカルプロセスとは違って扱う。そしてプロセスが移動する回数に上限を定めるための移動カウントパラメータを使用する。システム設計者の何人かは、プロセス移動は高価な操作であり、プロセスを余りにも頻繁に移動させるのを許すべきではないと感じている。そこで、この設計者のグループは、やり直しのできない移動方策を通常好む。すなわち、移動カウントの上限値は 1 に固定され、この方策ではプロセスは 1 回より多くは移動できない。しかし、別のシステム設計者は複数回数のプロセス移動 (特に長いプ

ロセス) はノードの動的な状態の変化に適合するのに大変役立つと感じている。したがって、この設計者のグループは移動カウント値の上限を 1 より大きい値 $k > 1$ に設定する。k の値は静的又は動的に決定されるであろう。その値は異なる特性をもつプロセスごとに異なるであろう。例えば、長いプロセス (実行時間の大きいプロセス) は短いプロセスに比べて、より多くの移動が許可される。

7.5　負荷共有方式

何人かの研究者は、システムのすべてのノードにおける作業負荷を等しくしようとする負荷バランスをとることは適切な目標ではないと信じている。その理由は、この目標を達成するために状態情報を収集するオーバヘッドは通常非常に大きい、特に多数のノードをもつ分散システムにおいて大きいからである。さらに、厳密な意味で、負荷バランスをとることは達成できない。その理由は、あるノードにおけるプロセスの数は常に変動しており、たとえ静的 (平均的) な負荷バランスが完全にとれていても、ノード間の一時的なアンバランスがどの瞬間でも存在しているからである。実際に、分散システムにおける資源の適切な使用のために、すべてのノードの負荷をバランスさせることは要求されていない。それよりも、他のノードに 2 つより多くのプロセスをもつ一方で、あるノードが空いているという状況を防止することが必要であり、またそれで十分でもある。それゆえ、この修正は動的負荷バランス (dynamic load balancing) という代わりに**動的負荷共有 (dynamic load sharing)** と呼ばれる。

7.5.1　負荷共有アルゴリズムの設計問題

負荷バランスアルゴリズムと同様に、**負荷共有アルゴリズム**の設計もまた、負荷推定方策、プロセス転送方策、状態情報交換方策、優先度割り当て方策、及び移動制限方策に関する適切な設計が要求される。しかし、負荷バランス法に比べ、負荷共有法の方がこれらのほとんどの方策を決定することは容易である。この理由は、負荷バランスアルゴリズムとは異なり、負荷共有アルゴリズムはシステムのすべてのノード上の平均作業負荷のバランスをとろうとはしないからである。その代わり、あるノードが重負荷のときに、空いているノードがないことを保証しようとする。負荷共有アルゴリズムに対する優先度割り当て方策及び移動制限方策は、負荷バランスアルゴリズムのものと同じである。したがって、それらの説明は繰り返さない。負荷共有法の上記以外の方策を以

下に述べる。

負荷推定方策

負荷共有アルゴリズムは、単にあるノードでプロセスが処理待ちのとき、空いているノードがないことを保証しようとするのみであるので、各ノードが処理中か空きかを知ることで十分である。したがって、負荷共有アルゴリズムは、ノードの上にいるプロセスの総数を計数するという最も単純な負荷推定方策を通常は採用する。

しかし前述のように、ノード上のプロセスの総数を計数する単純な負荷推定方策は現代の分散システムに使用するのに、空きノードの上でも幾つかのプロセスが常駐しているので、適していない。それゆえ、これらのシステムでは負荷推定方法として CPU 占有率を計測する方法が使用されるべきである。

プロセス転送方策

負荷共有アルゴリズムは通常、ノードの処理中か空きかについてのみ関心があるので、それらのほとんどは、**すべてか無 (all-or-nothing)** 方策を採用している。この方策は、すべてのノードの閾値が 1 に固定の単一閾値方策を使用する。すなわち、あるノードはプロセスをもっていないときのみ遠隔プロセスを受け入れる候補となり、プロセスを 1 つより多くもつと直ちにプロセスを転送する候補となる。Krueger 及び Livny [1987] は、すべてか無方策は次の意味で良くないことを指摘した。他のノードでプロセスが処理を待っていても、空きになるノードは実行するための新しいプロセスを直ちに獲得することができない、それによりシステムの使用可能な処理能力を損失する。彼らは、この損失を避けるために、現在空きではないが、間もなく空きになると予想されるノードに予測的な転送が必要であることを示した。この損失に対して、ある負荷共有アルゴリズムでは閾値を 1 の代わりに 2 を用いている。

もし CPU 占有率が負荷推定方策として使用されているならば、高低方策 (負荷バランス法のために論じた) がプロセス転送方策として使用されるべきであることに注意されたい。

配置方策

負荷共有アルゴリズムにおいて、**配置方策 (location policy)** は、負荷共有のためにシステム内で移動させられるプロセスの送信ノード又は受信ノードを決定する。転送するプロセスのための適切なノードを全体で探索するための起

動を行うノードのタイプによって、配置方策は次のタイプになる。

■ **送信者起動方策 (sender-initiated policy)**:この方策では、プロセスの送信者ノードがプロセスをどこに送信するのかを決定する。

■ **受信者起動方策 (receiver-initiated policy)**:この方策では、プロセスの受信者ノードがプロセスをどこから受信するかを決定する。

これらの方策のそれぞれの利点、欠点を以下に述べる。

送信者起動配置方策: **送信者起動配置方策**において、重負荷ノードは、作業を転送する先の軽負荷ノードを探索する。すなわち、この方法では、ノードの負荷が閾値より大きくなると、そのプロセスの 1 つ又はそれ以上を受け入れてくれる軽負荷ノードを見つけるために、メッセージをブロードキャストするか又は 1 つずつランダムに他のノードを探査する。あるノードへのプロセスの転送によって、その受信ノードの負荷の増加が閾値を越えないならば、そのノードは、送信ノードからのプロセスを受信するための有力な候補である。ブロードキャスト方法において、適切な受信ノードが存在するかしないかは、送信者ノードが他のノードからの応答メッセージを受信すると直ちに判明する。他方、ランダム探査の方法では、探査は、適切な受信ノードが見つかるまで、又は探査回数の上限 L_p に達するまで継続する。適切な受信ノードが見つからないなら、プロセスを生成したノードがそのプロセスを実行しなければならない。固定の回数制限をもつ探査の方法は、大規模なネットワークにおいてさえ負荷共有方策を実行するコストは膨大にはならないので、ブロードキャスト方法よりも大きい規模拡張性をもつ。Eager 他. [1986a] は彼らの分析を通して、この方策の性能は探査制限回数の選択に余り関係しないことを見つけた。小さい探査制限回数 (例えば、3 又は 5) の性能は大きい探査制限回数 (例えば 20) のものと同様であった。

受信者起動配置方策: **受信者起動配置方策**において、軽負荷ノードは、そこから作業が転送されるかもしれない重負荷ノードを探索する。すなわち、この方法では、ノードの負荷が閾値より小さくなると、実行のためのプロセスを受け入れる意志があることをブロードキャストするか、又は 1 つ又はそれより多くのプロセスを送り出すことのできる重負荷のノードを見つけるために 1 つずつランダムに他のノードを探査する。そのノードからのプロセスの転送により負荷が閾値よりも下回らないときに限り、そのノードはプロセスの 1 つを送り出す有力な候補である。ブロードキャスト方法において、受信ノードが他の

ノードから応答メッセージを受信すると直ちに適切なノードが見つけられる。他方、ランダム探査方法では、プロセスが得られるノードが見つかるまで、又は探査回数が静的制限回数 L_p に達するまで、探査が継続される。後者の場合、ノードは転送の起動を再び試みる前に、固定のタイムアウト期間の間待つ。

送信者起動アルゴリズムにおいて、スケジュールの決定は通常、プロセス到着時 (又はそのサブセット) になされる。他方、受信者起動アルゴリズムでは、スケジュールの決定は通常、プロセスの送出時 (又はそのサブセット) になされる。これによって、受信者起動方策は**先取り権付きプロセス移動** (preemptive process migration) 機能をサポートしないシステムとも動作できる。先取り権付きプロセス移動機能は、実行中のプロセスをあるノードから別のノードに転送することを認める。他方、**先取り権なしプロセス移動** (non-preemptive process migration) 機能のみをサポートするシステムにおいて、プロセスは実行開始前にのみ転送されうる。新しいノードへ送られるプロセスに付随していくプロセスの状態情報は、実行開始後には格段に複雑になるので、先取り権付きプロセス移動は、先取り権なしプロセス移動よりもコストの高いものとなる。受信者起動プロセス転送において、潜在送信者ノードに新しいプロセスが到着したとき、受信者ノードはその送信者と交渉を開始することは余りないので、受信者起動プロセス転送はほとんど先取り権付きである。しかし、送信者が起動したプロセス転送は、送信者が転送時にどちらを選ぶかによって、先取り権なしであるか、又は先取り権付きである。

Eager 他 [1986a] は 2 つの方策の性能を互いに、また負荷共有なしのシステムの性能と比較するために、単純な解析モデルを用いた。広いシステムパラメータに対して成り立つその結果を次に示す [Eager 他 1986a]。

■ 送信者起動及び受信者起動方策の両方は、負荷共有なしの方法に比べ、本質的な性能の改善がみられる。

■ 送信者起動方策は軽負荷から中程度の負荷に対して、受信者起動方式よりも好ましい。これは軽負荷システムにおいて、すべての空いているノードは必至になって作業を求めるので、受信者起動方策は多数の探査メッセージを生成するからである。

■ 受信者起動方策は高いシステム負荷において好ましいが、2 つの方策の下でのプロセス転送コストが同等である場合に限る。これは、受信者起動方策はクリティカルな時点 (システムの負荷が高いとき) で、システムに余分の負荷を架さないからである。しかし、システムが重負荷にあり、探査メッセージによ

る余分な負荷にほとんど対応できない正にそのとき、送信者発方策は多数の探査を生成する。

■ もし受信者起動方策の下のプロセス転送のコストが、先取り権付きプロセス転送のゆえに送信者起動方策よりも大きいならば、送信者起動方策はよりよい性能を提供する。

状態情報交換方策

負荷共有アルゴリズムはすべてのノードの平均負荷を等価しようとするものではないので、ノードが互いに周期的に状態情報を交換することは必要ではない。それより、ノードが軽負荷又は過負荷になるときのみ他のノードの状態を知る必要がある。それゆえ、負荷共有アルゴリズムにおいて、あるノードは通常、状態が変化するときのみ、他のノードと状態情報の交換を行う。この目的のために、2 つのよく使用される方策を以下に示す。

状態変化時ブロードキャスト: この方法では、ノードが軽負荷又は過負荷のどちらかになると**状態情報要求 (StateInformationRequest** メッセージをブロードキャストする。明らかに、送信者起動方策では、ノードが過負荷になるときのみ、このメッセージをブロードキャストする。受信者起動方策では、ノードが軽負荷になるときのみ、このメッセージをブロードキャストする。固定の閾値の値が 1 である受信者起動方策において、状態情報交換のこの方法は、**空き時ブロードキャスト (broadcast-when-idle) 方策**と呼ばれる。

状態変化時ポーリング: ブロードキャストプロトコルを用いる機構は大規模ネットワークに適していないので、ポーリング機構が通常、このようなシステムに使用される。この方法では、ノードの状態が変化するとき、他のすべてのノードと状態情報を交換することはせず、ランダムにポーリングを行い、相手のノードと状態情報を 1 対 1 で交換する。状態情報交換過程は、探査を行っているノードと負荷共有すべき適切なノードが見つかったとき、又は探査の制限回数に達したときに終了する。明らかに、送信者起動方策では、ノードが過負荷になるとき、ポーリングが行われる。固定の閾値が 1 である受信者起動方策では、この状態情報交換の方法は、空き時ポール (poll-when-idle) 方策と呼ばれる。

7.6　まとめ

分散オペレーティングシステムの資源管理者は、資源の使用、応答時間、ネットワーク輻輳、及びスケジュールオーバヘッドの組み合わせを最適化できるように、分散システムにおけるプロセスを1つ以上又は1まとまりの資源プールに割り付けするスケジュールを行う。プロセスのスケジュールの決定は、プロセスの資源要求、システムの異なるノードにおける様々な資源の可用性などの要素に基づく。良好な全体スケジューリングアルゴリズムは、プロセスに関する事前の知識を要しないこと、動作が動的であること、迅速な決定能力、システム性能とスケジューリング効率のバランス、安定性、規模拡張性、フォールトトレラント性、及びサービスの公平性のような特性をもつべきである。全体スケジューリングアルゴリズムに用いられる3つの異なる方法は、タスク割り付け、負荷バランス方法、及び負荷共有方法である。

タスク割り付け方法において、プロセス割り付けの決定は基本的に実行されるプロセス及びシステム両者の特性の事前知識に基づいている。基本的タスク割り付けモデルは、プロセス間の通信コストを最小化するように、及び一連のタスクのターンアラウンド時間を改善するように、システムの様々なノードへのタスク割り付けを扱う。基本的タスク割り付けモデルへの拡張は、干渉コスト、優先度関係、メモリサイズの制約などの要素も考慮する。タスク割り付け方法は、スケジュールされるすべてのプロセスの特性があらかじめ既知であるという仮定で動作すること、及びタスク割り付けアルゴリズムは一般的に動作が動的でないことにより、その適用性に限界がある。

負荷バランス方法において、プロセス割り付けの決定は、システムのすべてのノードの平均作業負荷を等しくしようとする。動的負荷バランスアルゴリズムに伴う基本的問題は、使用される全体状態情報の量について決定することである。この面で、成功している方法は少ないが、その結果は、現在のシステム全体の状態をより正確に記述する大量の情報を収集しようとすることは必ずしも益にはならないことを示している。したがって、あるアルゴリズムにおける全体知識の程度は、何らかの方法で、アルゴリズムの複合的性能目標に対して正規化されなければならない。この全体知識と目標との関係及び正規化が現在の研究の課題である。

最後に、負荷共有方法において、プロセス割り付け決定は、すべてのノードに対し、システムに十分なプロセスが存在するなら、すべてのノードを使用中にしようとする。これは、他のノードに処理を待っているプロセスが存在する

ならば、どのノードも空きではないことを保証することによって達成される。負荷共有アルゴリズムはすべてのノードの負荷のバランスをとることは行わないので、負荷バランスアルゴリズムに比較して実装が簡単で容易である。しかし、数多くの全体スケジューリングアルゴリズムに対して、「アルゴリズム A は B よりもすぐれているか」という質問に対する絶対的な答えは存在しない。それゆえ、全体スケジューリングに含まれるプロセスのより良い理解を得ることが、この種のアルゴリズムの研究の狙いであるべきである。

7.7　演習問題

7.1 全体スケジューリングアルゴリズムに対し、最適方法よりも準最適結果をもたらす発見的方法が一般的に好ましい理由は何か。

7.2 「プロセッサスラッシング」とは何か。プロセッサスラッシングになりうる 2 つの全体スケジューリングアルゴリズムの例を挙げよ。この問題を扱うためにとられる必要な対策を挙げよ。

7.3 分散システムにおけるプロセスのタスクをスケジューリングするためのタスク割り付け方策に基づきスケジューリングアルゴリズムを設計することを想定する。プロセスのタスクに対してどのようなタイプのコスト情報を必要とするか。これらのコストの概略の推定を行う方法を提案せよ。

7.4 システムは 3 つのプロセッサ p_1、p_2、及び p_3 からなり、4 つのタスク t_1、t_2、t_3、及び t_4 をもつ 1 つのプロセスがこのシステムで実行される。E_{ij} はタスク t_1 をプロセッサ p_j で実行するコスト、及び C_{ij} はタスク t_i 及び t_j が異なるプロセッサに割り付けられたときの両者の間の通信コストとする。$E_{11} = 31$, $E_{12} = 4$、$E_{13}14$、$E_{21} = 1$、$E_{22} = 5$、$E_{23} = 6$、$E_{31} = 2$、$E_{32} = 4$、$E_{33} = 24$、$E_{41} = 3$、$E_{42} = 28$、$E_{43} = 10$、$C_{12} = 35$、$C_{13} = 3$、$C_{14} = 8$、$C_{23} = 6$、$C_{24} = 4$、及び $C_{34} = 23$ とする。プロセッサへのタスクの最適割り付けを見出せ、また最適割り付けのコストを計算せよ。次に、t_1 及び t_2 が p_1 に、t_3 が p_2 に、t_1 が p_3 に割り付けられる場合のコストと比較せよ。

7.5 システムは 2 つのプロセッサ p_1 及び p_2 からなり、6 つのタスク t_1、t_2、t_3、t_4、t_5、及び t_6 をもつ 1 つのプロセスがこのシステムで実行される。E_{ij} はタスク t_1 をプロセッサ p_j で実行するコスト、C_{ij} はタスク t_i 及び t_j が異なるプロセッサに割り付けられたときの両者の間の通信コスト、及び I_{ij} はタスク t_i 及び t_j が同一のプロセッサに割り付けられたときの両者の間の干渉コスト (interference cost) とする。干渉コストは 2 つのタスク間の非互換性の程度

を反映し、2つのタスクが同一のプロセッサに割り付けられた場合に生じる[Lo 1988]。$E_{11}=20$, $E_{12}=50$、$E_{21}=25$、$E_{22}=10$、$E_{31}=5$、$E_{32}=20$、$E_{41}=10$、$E_{42}=20$、$E_{51}=10$、$E_{52}=20$、$E_{61}=50$、$E_{62}=10$、$C_{12}=15$、$C_{23}=50$、$C_{34}=15$、$C_{45}=50$、及び $C_{56}=15$ とする。他のすべての C_{ij} の値はゼロである。さらに、いずれのタスク間の干渉コストも10(すなわち、$I_{ij}=10$、もし i が j に等しくないならば)とする。次の場合の割り付けコストを求めよ。

(a) タスク t_1 は p_1 に割り付けられている。他のすべてのタスクは p_2 に割り付けられている。干渉コストを考慮しない場合。

(b) タスクセット (t_1、t_2、t_3) 及び (t_4、t_5、t_6) がそれぞれ p_1 及び p_2 に割り付けられている。干渉コストを考慮しない場合。

(c) タスク t_1 が p_1 に割り付けられている。他のすべてのタスクが p_2 に割り付けられている。干渉コストを考慮する場合。

(d) タスクセット (t_1、t_2、t_3) 及び (t_4、t_5、t_6) がそれぞれ p_1 及び p_2 に割り付けられている。干渉コストを考慮する場合。

これらの4つの場合の割り付けコストからどのような結論が得られるか。

7.6 あるコンピュータシステムは3つのプロセッサ p_1、p_2、及び p_3 をもっている。そして同じタイプのプロセスを処理するのに使用される。すべてのプロセスは6つのタスク t_1、t_2、t_3、t_4、t_5、及び t_6 をもっている。プロセスの到着率は決定論的な過程であると仮定する。各タスクの実行時間は指数分布に従うものとし、タスク t_1、t_2、t_3、t_4、t_5、及び t_6 の平均実行時間はそれぞれ、1、10、1、10、1、及び10とする。タスクの間には強い先行関係があり、あるプロセスのタスク t_j は、同じプロセスのタスク t_i(i < j) の実行が完了したときにのみ実行される。このシステムで実行されるプロセスに対して、次の割り付けのどれが最善の応答時間となり、どれが最悪の応答時間となるか。

(a) タスク対 (t_1、t_2)、(t_3、t_4)、及び (t_5、t_6) がそれぞれプロセス p_1、p_2、及び p_3 に割り付けられる。

(b) タスク対 (t_1、t_6)、(t_2、t_3)、及び (t_4、t_5) がそれぞれプロセス p_1、p_2、及び p_3 に割り付けられる。

(c) タスク対 (t_1、t_4)、(t_2、t_5)、及び (t_3、t_6) がそれぞれプロセス p_1、p_2、及び p_3 に割り付けられる。

また、その答えの理由を述べよ。

7.7 システムは2つのプロセッサ p_1 及び p_2 をもち、p_1 は限られたメモリ容量をもち、p_2 は無制限のメモリ容量をもっている。複数のタスクを持つプロセ

スがこのシステムで実行されるとする。各プロセッサ上の各タスクに対する実行コスト、タスク間の通信コスト、各タスクの必要メモリ量、及びプロセッサ p_1 のメモリ容量が与えられる。a_1 がメモリサイズの拘束条件なしに実行時間及び通信コストの総計を最小にする割り付けであるとし、a_2 はメモリサイズの拘束条件の下に実行時間及び通信コストの総計を最小にする割り付けであるとする。a_2 によって p_1 に割り付けられるタスク群は、a_1 によって p_1 に割り付けられるタスク群のサブセットであることを証明せよ。

7.8 分散システムに対するスケジューリング方式としてタスク割り付け方法の実際的な適用性について意見を述べよ。

7.9 次のタイプの分散システムに対するスケジューリング方式として負荷バランス方法の実際的な適用性について意見を述べよ。

(a)LAN ベースの分散システム

(b)WAN ベースの分散システム

(c) プロセッサプールモデルに基づく分散システム

(d) ワークステーションサーバモデルに基づく分散システム

7.10 分散オペレーティングシステムの一人の設計者は、分散システムにおいて状態情報を収集することは高いコストにつくという意見をもっている。それゆえに、プロセッサプールモデルに基づく分散システムに対して、その設計者は次のプロセス配置ルールを使用する負荷バランス方法を使用することを決定する。「端末 i から生成するすべてのプロセスはプロセッサ j で実行する ($j \ll i$)。j の値は、i のすべての値に対して定義され、i の幾つかの値に対して、j の値は同じこともある。」

このシステムに対して、スケジューリングアルゴリズムの設計者の選択は適切であるか読者の意見及びその理由を述べよ。あるとすれば、このスケジューリングアルゴリズムのもつ欠点は何か。もし読者がこのアルゴリズムがこのシステムに対して適切でないと判断するならば、このシステムに対して適切な全体スケジューリングアルゴリズムを提案せよ。

7.11 分散システムにおけるプロセスの全体スケジューリングのための集中型負荷バランスアルゴリズムを設計する場合を想定する。取り扱わなければならない問題は何か。またその問題のそれぞれを取り扱う適切な方法を提案せよ。

7.12 分散システムのための負荷バランスアルゴリズムを設計する場合を想定する。分散システムが次のモデルに基づくならば、そのアルゴリズムには、利己的、利他的、及び中間的優先度割り当て方策のいずれを読者は使用するか。

(a) ワークステーションサーバモデル

(b) プロセッサプールモデル

また、その理由を述べよ。

7.13 分散システムにおける負荷バランスアルゴリズムのプロセッサ転送方策として、高低方策を使用することを決定したと想定する。この実装において、高マーク及び低マークの値を選ぶための適切な方法を提案せよ。これらの閾値はシステムにおけるすべてのプロセッサに対して同じでなければならないか。その答えの理由を述べよ。

7.14 分散システムにおいて、厳密な意味での負荷バランスは達成できない。これを論じよ。

7.15 分散システムにおいて、プロセススケジューリングのための負荷バランス法と負荷共有法との主な相違点は何か。2 つの方法の実装において使用される様々な方策のうちで、どの方策が異なり、どの方策が同じか。

7.16 読者が分散システムの負荷共有アルゴリズムを設計する場合を想定する。そのアルゴリズムにおいて、読者は送信者起動方策と受信者起動方策のどちらを選択するか。また、その理由を述べよ。

7.17 負荷バランスアルゴリズムにおいて負荷推定に使用され得る方策を提案せよ。提案された方策の相対的な利点及び欠点を論じよ。またそれらの方策のうちで、負荷共有アルゴリズムにおいても使用できる方策はどれか。もしなければ、負荷共有アルゴリズムのための適切な負荷推定の方策を提案せよ。

7.18 システムが 2 つのプロセッサ p_1 及び p_2 をもっている。ある特定の瞬間において、p_1 には、200 秒のサービス時間が残っている 1 つのプロセスがあり、p_2 には、1 秒のサービス時間が残っている 100 のプロセスがある。このとき、1 つの新しいプロセスがシステムに入ってくることを想定する。次の場合の新しいプロセスの応答時間を計算せよ。

(a) 新しいプロセスは 1 秒のプロセスで、実行のために p_1 に配置される。

(b) 新しいプロセスは 1 秒のプロセスで、実行のために p_2 に配置される。

(c) 新しいプロセスは 200 秒のプロセスで、実行のために p_1 に配置される。

(d) 新しいプロセスは 200 秒のプロセスで、実行のために p_2 に配置される。

システムに他に新しい到着がないと仮定せよ。得られた結果から、負荷バランスアルゴリズムにおいて使用される負荷推定方策について、どのような結論を得ることができるか。

7.19 ある分散システムは、先取り権付きのプロセス移動機能をサポートしていない。このシステムにおいて、プロセスのスケジューリングのために負荷共有アルゴリズムを読者が設計する場合を想定する。そのアルゴリズムで読者は

送信者起動方策又は受信者起動方策のいずれを使用するか。また、その理由は何か。

7.20 ある分散システムは、負荷共有アルゴリズムのためのプロセス転送方策として、すべてか無方策を使用している。このシステムのプロセッサの処理能力がなぜ適切に使用されないのかを説明せよ。この問題を克服する適切な方法を提案せよ。

7.21 分散システムにおけるプロセスの全体スケジューリングの分野で、さらに注意を読者が必要と考える研究事項は何か。

参考文献

[**Ahrens and Hansen 1995**] Ahrens, J. P., and Hansen, C. D., "Cost-Effective Data-Parallel Load Balancing," In: Proceedings of the 24th Annual International Conference on Parallel Processing, CRC Press, Boca Raton, FL (August 1995).

[**Alonso and Cova 1988**]**A** lonso, R., and Cova, L. L., "Sharing Jobs Among Independently Owned Processors," In: Proceedings of the 8th International Conference on Distributed Computing Systems, IEEE, New York, pp. 282-288 (June 1988).

[**Arora and Rana 1980**] Arora, R. K., and Rana, S. P., "Heuristic Algorithms for Process Assignment in Distributed Computing Systems," Information Processing Letters, Vol. 11, No. 4/5, pp. 199-203. (1980).

[**Atallah et al. 1992**] Atallah, M. J., Black, D. C., Marinescu, D. C., Seigel, H. J., and Casavant, T. L., "Models and Algorithms for Co-Scheduling Computer Intensive Tasks on a Network of Workstations," Journal of Parallel and Distributed Computing, Vol. 16, pp. 319-327 (1992).

[**Barak and Shiloh 1985**] Barak, A., and Shiloh, A., "A Distributed Load Balancing Policy for a Multicomputer," Software—Practice and Experience, Vol. 15, No. 9, pp. 901-903 (1985).

[**Bokhari 1979**] Bokhari, S. H., "Dual Processor Scheduling with Dynamic Reassignment," IEEE Transactions on Software Engineering, Vol. SE-5, No. 4, pp. 341-349 (1979).

[**Bokhari 1981a**] Bokhari, S. H., "On the Mapping Problem," IEEE Transactions on Computers, Vol. C-30, No. 3, p. 207-214 (1981).

[Bokhari 1981B] Bokhari, S. H., "A Shortest Tree Algorithm for Optimal Assignments Across Space and Time in a Distributed Processor System," IEEE Transactions on Software Engineering Vol. SE-7, No. 6, pp. 583-589 (1981).

[Bonomi and Kumar 1990] Bonomi, F., and Kumar, A., "Adaptive Optimal Load Balancing in a Heterogeneous Multiserver System with a Central Job Scheduler," IEEE Transactions on Computers, Vol. C-39, pp. 1232-1250 (1990).

[Bryant and Finkel 1981] Bryant, R. M., and Finkel, R. A., "A Stable Distributed Scheduling Algorithm," In: Proceedings of the 2nd International Conference on Distributed Computing Systems, IEEE, New York, pp. 314-323 (April 1981).

[Casavant and Kuhl 1986] Casavant, T. L., and Kuhl, J. G., "A Formal Model of Distributed Decision Making and Its Application to Distributed Load Balancing," In: Proceedings of the 6th International Conference on Distributed Computing ASAystems, IEEE , New York, pp. 232-239 (May 1986).

[Casavant and Kuhl 1988a] Casavant, T. L., and Kuhl, J. G., "A Taxonomy of Scheduling in General Purpose Distributed Computing System," IEEE Transactions on Software Engineering, Vol. SE-14, No. 2, pp.141-154(1988).

[Casavant and Kuhl 1988b] Casavant, T. L., and Kuhl, J. G., "Effects of Response and Stability on Scheduling in Distributed Computing Systems," IEEE Transactions on Software Engineering, Vol. SE-14, No. 11, pp. 1578-1587 (1988).

[Chang and Oldham 1995] Chang, H. W. D., and Oldham, W. J. B., "Dynamic Task Allocation Models for Large Distributed Computing Systems," IEEE Transactions on Parallel and Distributed Systems, Vol. 6, No. 12, pp. 1301-1315 (1995).

[Chou and Abraham 1982] Chou, T. C. K., and Abraham, J. A., "Load Balancing in Distributed Systems," IEEE Transactions on Software Engineering, Vol. SE-8, No. 4,pp. 401-412 (1982).

[Chou and Abraham 1983] Chou, T. C. K., and Abraham, J. A., "Load Distribution Under Failure in Distributed Systems," IEEE Transactions on Computers, Vol. C-32, No. 9, pp. 799-808 (1983).

[**Chu et al. 1980**] Chu, W., Holloway, L. J., Lan,M. T., and Efe K., "Task Allocation in Distributed Data Processing," IEEE Computer, Vol.13, pp.57-69(1980).

[**Chu and Lan 1987**] Chu, W. W., and Lan, L. M-T., "Task Allocation and Precedence Relations for Distributed Real-Time Systems," IEEE Transactions on Computers, Vol. C-36, No. 6, pp. 667-679 (1987).

[**Clark and McMillin 1992**] Clark, H., and McMillin, B., "DAWGS—A Distributed Compute Server Utilizing Idle Workstations," Journal of Parallel and Distributed Computing, pp. 175-186 (February 1992).

[**Dandamudi 1995**] Dandamudi, S., "Performance Impact of Scheduling Discipline on Adaptive Load Sharing in Homogeneous Distributed Systems" In: Proceedings of the 15th International Conference on Distributed Computing Systems IEEE, New York (1995).

[**Eager et al. 1986a**] Eager, D. L., Sazowaska, E. D., and Zahorjan, J., "A Comparison of Receiver-Initiated and Sender- Initiated Adaptive Load Sharing," Performance Evluation, Vol. 6, pp. 53-68 (1986).

[**Eager et al. 1986b**] Eager, D. L., Lazowska, E. D., and Zahorjan, J., "Adaptive Load Sharing in Homogeneous Distributed Systems." IEEE Transactions on Software Engineering, Vol. SE-12, No. 5, pp. 662-675 (1986).

[**Efe 1982**] Efe, K., "Heuristic Models of Task Assignment Scheduling in Distributed Systems," IEEE Computer, Vol. 15, pp. 50-56 (1982).

[**Efe and Groselj 1989**] Efe, K., and Groselj, B., "Minimizing Control Overheads in Adaptive Load Sharing," In: Proceedings of the 9th International Conference on Distributed Computing Systems, IEEE, New York, pp. 307-315 (June 1989).

[**Efe and Krishnamoorthy 1995**] Efe K., and Krishnamoorthy, V., "Optimal Scheduling of Compute-Intensive Tasks on a Network of Workstations," IEEE Transactions on Parallel and Distributed Systems, Vol. 6, No. 6, pp. 668-673 (1995).

[**Efe and Schaar 1993**] Efe, K., and Schaar, M., "Performance of Co-scheduling on a Network of Workstations," In: Proceedings of the 13th International Conference on Distributed Computing Systems,k IEEE, New York, pp. 525-531 (1993).

[El-Rewini et al. 1994] El-Rewini, H., Lewis, T., and Ali, H., Task Scheduling in Parallel and Distributed Systems, Prentice-Hall, Englewood Cliffs, NJ (1994).

[El-Rewini et al. 1995] El-Rewini, H., Lewis, T., and Ali, H., "Task Scheduling in Multiprocessing Systems," IEEE Computer, Vol. 28, No.12, pp.27-37(December 1995).

[Ferguson et al. 1988] Ferguson, D., Yemini, Y., and Nikolaou, C., "Microeconomic Algorithms for Load Balancing in Distributed Computer Systems," In: Proceedings of the 8th International Conference on Distributed Computing Systems, IEEE New York, pp. 491-499 (1988).

[Gao et al. 1984] Gao, C., Liu, J. W. S., and Railey, M., "Load Balancing Algorithms in Homogeneous Distributed Systems," In: Proceedings of the 1984 International Conference on Parallel Processing, CRC Press, Boca Raton, FL, pp. 302-306 (August 1984).

[Hagman 1986] Hagman, R., "Process Server: Sharing Processing Power in a Workstation Environment," In: Proceedings of the 6th International Conference on Distributed Computing Systems, IEEE, New York, pp.260-267(1986).

[Heirich and Taylor 1995] Heirich, A., and Taylor, S., "A Parabolic Load Balancing Method," In: Proceedings of the 24th Annual International Conference on Parallel Processing, CRC Press, Boca Raton, FL (August 1995).

[Hsu and Liu 1986] Hsu, C. H., and Liu, J. W. S., "Dynamic Load Balancing Algorithm in Homogeneous Distributed Systems," In: proceedings of the 6th International Conference on Distributed Computing Systems, IEEE, New York, pp. 216-222 (May 1986).

[Hwang et al 1982] Hwang, K., Croft, W. J., Goble, G. H., Wah, B. W., Briggs, F. A., Simmons, W. R., and Coates, C. L., "A UNIX-Based Local Computer Network with Load Balancing," IEEE Computer, Vol. 15, pp.55-66(April 1982).

[Kleinrock and Korfhage 1989] Kleinrock, L., and Korfhage, W., "Collecting Unused Processing Capacity: An Analysis of Transient Distributed System," In: Proceedings of the 9th International Conference on Distributed Computing Systems, IEEE, New York, pp. 482-489 (June 1989).

[**Krueger and Chawla 1991**] Krueger, P., and Chawla, R., "The Stealth Distributed Scheduler," In: Proceedings of the 11th International Conference on Distributed Computing Systems, IEEE, New York, pp. 336-343 (1991).

[**Krueger and Livny 1987**] Krueger, P.,k and Livny, M., "The Diverse Objectives of Distributed Scheduling Policies," In: Proceedings of the 7th International Conference on Distributed Computing Systems, IEEE, New York, pp. 242-249(September 1987).

[**Kunz 1991**] Kunz, T., "The Influence of Different Workload Descriptions on a Heuristic Load Balancing Scheme," IEEE Transactions on Software Engineering, Vol. SE-17, No. 7, pp. 725-730 (1991).

[**Lee and Towsley 1986**] Lee, K. J., and Towsley, D., "A Comparison of Priority-Based Decentralized Load Balancing Policies," In: Proceedings of the 10th Symposium on Operating System Principles, Association for Computing Machinery, New York, NY, pp. 70-77 (November 1986).

[**Lin and Raghavenda 1992**] Lin, H. C., and Raghavendra, C. S., "A Dynamic Load-Balancing Policy with a Central Job Dispatcher (LBC)," IEEE Transactions on Software Engineering, Vol. 18, No. 2, pp. 148-158 (1992).

[**Litzkow et al. 1988**] Litzkow, M. J., Livnhy, M., and Mukata, M. W., "Condor—A Hunter of Idle Aworkstations," In: Proceedings of the 8th International Conference on Distributed Computing Systems, IEEE, New York, pp. 104-111 (June 1988).

[**Livny and Melman 1982**] Livny, M., and Melman, M., "Load Balancing in Homogeneous Broadcast Distributed Systems," In: Proceedings of the ACM Computer Network Performance Symposium, pp., 47-55 (April 1982).

[**Lo 1988**] Lo, V. M., "Heuristic Algorithms for Task Assignment in Distributed Systems," IEEE Transactions on Computers, Vol. 37, No. 11, pp. 1384-1397 (1988).

[**Malone et al. 1988**] Malone, T. W., Fikes, R. E., Grant, K. R., and Howard, M. T., "Enterprise: A Market-like Task Scheduler for Distributed Computing Environments," The Ecology of Computation, Huberman, B. A., Ed., North-Holland, Amsterdam, pp. 177-205 (1988).

Mehra and Wah 1995a] Mehra, P., and Wah, B., "Synthetic Workload Generation for Load-Balancing Experiments," IEEE Parallel and Distributged Technology, Systems and Applications, pp. 4-19 (Fall 1995).

[Mehra and Wah 1995b] Mehra, P., and Wah, B., Load Balancing: An Automated Learning Approach, World Scientific, River Edge, NJ (1995).

[Milojcic 1994] Milojcic, D. S., Load Distribution, Implementation for the Mach Microkernel, Verlag Vieweg, Wiesbaden (1994).

[Mirchandaney et al. 1989a] Mirchandaney, R., Towsley, D., and Stankovic, J. A., "Adapve Load Sharing in Heterogeneous Systems," In: Proceedings of the 9th International Conference on Distributed Computing Systems, IEEE New York, NY, pp. 298-306 (June 1989).

[Mirchndaney et al. 1989b] Mirchandaney, R., Towsley, D., and Stankovic, J. A., "Analysis of the Effects of Delays on Load Sharing," IEEE Transactions on Computers, Vol. 38, pp. 1513-1525 (1989).

[Mukta and Livny 1987] Mukta, M. W., and Livny, M., "Scheduling Remote Processing Capacity in a Workstation-Processor Bank Computing Systems," In: Proceedings of the 7th International Conference on Distributed Computing Systems, IEEE, New York, NY, pp. 2-9 (September 1987).

[Ni and Hwang 1985] Ni, L. M., and Hwang, K., "Optimal Load Balancing in a Multiple Processor System with Many Job Classes," IEEE Transactions on Software Engineering, Vol. SE-11, No. 5, pp. 491-496 (1985).

[Ni et al. 1985] Ni, L. M., Xu, C. W., and Gendreau, T. B., "A Distributed Drafting Algorithm for Load Balancing," IEEE Transactions on Software Engineering, Vol, SE-11, No. 10, pp. 1153-1161 (1985).

[Nichols 1987] Nichols. D. A., "Using Idle Workstations in aShared Computing Environment," In: Proceedings of the 11th Symposium on Operating System Principles, Association for Computing Machinery, New York, NY, pp. 5-12, (November 1987).

[Pulidasa et al. 1988] Pulidas, S., Towsley, D., and Stankovic, J. A., "Imbedding Gradient Estimators in Load Balancing Algorithms," In: Proceedings of the 8th International Conference on Distributed Computing Systems, IEEE, New York, NY, pp. 482-490 (1988).

[Rao et al. 1979] Rao, G. S., Stone, H. S., and Hu, T. C., "Assignment of Tasks in a Distributed Processor System with Limited Memory," IEEE Transactions on Computers, Vol. C-28, No. 4, pp. 291-299 (1979).

[Shen and Tsai 1985] Shen, C. C., and Tsai, W. H., "A Graph Matching Approach to Optiomal Task Assignment in Distributed Computing Systems with Minimax Criterion," IEEE Transactions on Computers, Vol.C-34, pp.197-203(1985).

[Shirazi et al. 1995] Shirazi, B. A., Hurson, A. R., and Kave, K. M. (Eds.), Scheduling and Load Balancing in Parallel and Distributed Systems, IEEE Computer Society Press, Los Alamitos, CA (1995).

[Shivaratri and Krueger 1990] Shivaratri, N. G., and Krueger, p., "Two Adaptive Location Policies for Global Scheduling Algorithms," In: Proceedings of the 10th International Conference on Distributed Systems, IEEE, New York, NY, pp. 502-509 (1990).

[Shivaratri et al. 1992] Shivaratri, N. G., Krueger, P., and Singhal, M., "Load Distributing for Locally Distributed Systems," IEEE Computer, Vol. 25, No. 12, pp. 33-44 (1992).

[Singhal and Shivaratri 1994] Singhal, M., and Shivaratri, N. G., Advanced Concepts in Operating Systems, McGraw-Hill, New York (1994).

[Smirni et al. 1995] Smirni, E., Rosti, E., and Serrazi, G., "performance Gains from Leaving Idle ProCessors in Multiprocessor Systems," In: Proceedings of the 24th Annual International Conference on Parallel Processing, CRC Press, Boca Raton, FL (August 1995).

[Smith 1980] Smith, R. G., "The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem-Solver," IEEE Transactions on Computers, Vol. C-29, No. 12, pp. 1104-1113 (1980).

[Srinivasan and Jha 1995] Srinivasan, S., and Jha, N. K., "Task Allocation for Safety and Reliablility in Distributed Systems," In: Proceedings of the 24th Annual International Conference on Parallel Processing, CRC Press, Boca Raton, FL (August 1995).

[Stankovic 1984] Stankovic, J. A., "Simulation of Three Adaptive, Decentralized Controlled, Job Scheduling Algorithms," Computer Networks, Vol. 8, No. 3, pp. 199-217 (1984).

[Stankovic 1985] Stankovic, J. A., "Stability and Distributed Scheduling Algorithms," IEEE Transactions on Software Engineering, Vol. SE-11, No.10, pp.1141-1152 (1985).

[Stankovic and Sidhu 1984] Stankovic, J. A., and Sidhu, I. S., "An Adaptive Bidding Algorithm for Processses, Clusters and Distributed

Groups," In: Proceedings of the 4th International Conference on Distributed Computing Systems, IEEE, New York, NY, pp. 49-59 (May 1984).

[Stone 1977] Stone, H. S., "Multiprocessor Scheduling with the Aid of Network Flow Algorithms," IEEE Transactions on Software Engineering, Vol. SE-3, No. 1, pp. 85-93 (1977).

[Stone 1978] Stone, H. S., "Critical Load Factors in Two-Processor Distributed Systems," IEEE Transactions on Software Engineering, Vol. SE-4, No. 3, pp. 254-258 (1978).

[Stumm 1988] Stumm, M., "The Design and Implementaton of a Decentralized Scheduling Facility for a Workstation Cluster," In: Proceedings of the 2nd IEEE Conference on Computer Workstations, IEEE, New York, NY, pp. 12-22 (March 1988).

[Svensson 1990] Svensson, A., "History—An Intelligent Load Sharing Filter," In: Proceedings of the 10th International Conference on Distributed Computing Systems, IEEE, New York, NY, pp. 546-553 (1990).

[Tanenbaum 1995] Tanenbaum, A. S., Distributed Operating Systems, Prentice-Hall, Englewood Cliffs, NJ (1995).

[Tantawi and Towsley 1985] tantawi, A. N., and Towsley, D., "Optimal Stataic Load Balancing in Distributed Computer Systems," Journal of ACM, Vol. 32, No. 2, pp.445-465 (1985).

[Theimer and Lantz 1989] Theimer, M. M., and Lantz, K. A., "Finding Idle Machines in a Workstaiton-Based Distibuted System," IEEE Transactions on Software Engineering, Vol. 15, pp. 1444-1458 (1989).

[Tilborg and Wittie 1984] Tilborg, A. M., and Wittie, L. D., "Wave Scheduling—Decentralized Scheduling of Task Forces in Multicomputers," IEEE Transactions on Computers, Vol. C-33, pp. 835-844 (1984).

[Wah 1984] Wah, B. W., "A Comparative Study of Distributed Resource Sharing on Multiprocessors," IEEE Transactions on Computers, Vol. C-33, pp. 700-711 (1984).

[Wah and Juang 1985] Wah, B. W., and Juand, J-Y, "Resource Scheduling for Local Computer Systems with a Multiaccess Network," IEEE Transactions on Computers, Vol. C-34, No. 2, pp. 1144-1157 (1985).

[Waldspurger et al. 1992] Waldspurger, C. A., Hogg, T., Huberman, B. A., Kephart, J. O., and Stornetta, W. S., "Spawn: A Distributed Computational Economy," IEEE Transactions on Software Engineering, Vol. 18, No. 2, pp. 103-117 (1992).

[Wang and Morris 1985] Wang, Y. T., and morris, R. J. T., "Load Sharing in Distributed Systems," IEEE Transactions on Computers, Vol. C-34, No. 3, pp. 204-217 (1985).

[Wu and Liu 1980] Wu, C. S., and Liu, M. T., "Assignment of Tasks and Resources for Distributed Processing," In: Proceedings of the 1980 International Conference on Computers and Communications (COMPCON-80), IEEE, New York, NY, pp. 655-662 (Fall 1980).

[Zhou 1988] Zhou, S., "A Trace-Driven Simulation Study of Dynamic Load Balancing," IEEE Transactions on Software Engineering, Vol. 14, pp. 1327-1341 (1988).

インターネット参考文献

本章に関係する他のトピックスに関する参考文献は、以下のところで見つけることができる。

ftp:ftp.cs.umanitoba.ca/pub/bibliographies/Parallel/Load.Balance1.html
ftp:ftp.cs.umanitoba.ca/pub/bibliographies/Parallel/Load.Balance2.html
ftp:ftp.cs.umanitoba.ca/pub/bibliographies/Parallel/load.Balance.5.html
ftp:ftp.cs.umanitoba.ca/pub/bibliographies/Parallel/scheduling.html
ftp:ftp.cs.umanitoba.ca/pub/bibliographies/Distributed/dshell.html

第8章

プロセス管理

8.1 はじめに

通常の (集中型) オペレーティングシステムでは、**プロセス管理 (process management)** は、システムのプロセッサを、すべてのプロセスの間で、共有するための機構及び方策を扱う。同様に、分散オペレーティングシステムでは、プロセス管理の主な目標は、システム全体の処理資源を、すべてのプロセスの間で、共有することにより、それらを可能な限り最善に使用するようにすることである。この目標を達成するために、3つの重要な概念が分散オペレーティングシステムで使用される。

1. **プロセッサ配置 (processor allocation)** は、どのプロセスがどのプロセッサに割り当てられるべきかを決定する過程を扱う。
2. **プロセス移動 (process migration)** は、プロセスが現在の場所から割り当てられたプロセッサへ移動することを扱う。
3. **スレッド (thread)** はシステムの処理能力のより良い使用のために、細分された並列処理を扱う。

プロセッサ配置の概念を資源管理に関する前章で既に述べた。それゆえ、本章はプロセス移動及びスレッドの概念について述べる。

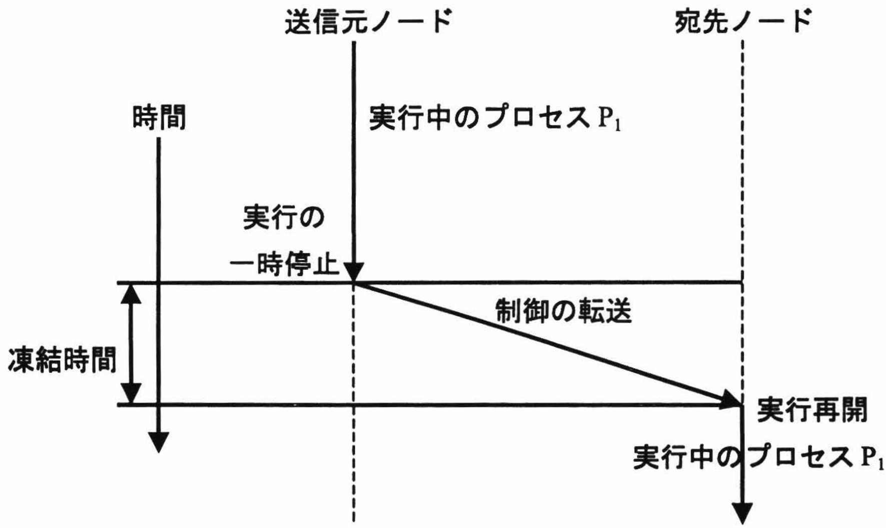


図 **8.1**　移動するプロセスの実行の流れ

8.2　プロセス移動

プロセス移動 (process migration) は、プロセスの現在位置 (送信元ノード (source node)) から別のノード (宛先ノード (destination node)) へのプロセスの再配置である。移動プロセスの実行のフローを図 8.1 に示す。

プロセスは送信元ノードでの実行開始以前に、又は実行の途中で移動させられるかもしれない。前者は**先取り権なし (non-preemptive) プロセス移動**として知られ、後者は先取り権付き (preemptive) **プロセス移動**として知られている。既に実行中のプロセスではプロセス環境も新しいノードへのプロセスに付随していかなければならないので、先取り権付きプロセス移動は先取り権なしプロセス移動よりもコストがかかる。

プロセス移動は次の主なステップを含んでいる。

1. 移動させるプロセスの選択
2. 選択されたプロセスが移動される宛先ノードの選択
3. 選択されたプロセスの宛先ノードへの転送

最初の2つのステップはプロセス移動方策により、3番のステップはプロセス移動機構によってなされる。送信元ノード、宛先ノード、及び移動されるプロセスの選択方策については資源管理に関する前章にて既に述べた。本章では、現存の分散オペレーティングシステムによって使用されているプロセス移動機構について述べる。

8.2.1 良好なプロセス移動機構の望ましい特性

良好なプロセス移動機構は、透過性 (transparency)、最小の干渉 (minimal interference)、最小の残留依存性 (minimal residue dependency)、効率 (efficiency)、堅固性 (robustness)、及びプロセス間の通信 (communication between coprocesses) の特性をもつべきである。

透過性

透過性はプロセス移動をサポートするシステムにとって重要な要件である。透過性のレベルには次のものがある。

1. **オブジェクトアクセスレベル**。オブジェクトアクセスレベルでの透過性は、先取り権なしのプロセス移動機能をシステムがサポートするための最小限の要件である。システムがオブジェクトアクセスレベルで透過性をサポートするならば、ファイルや装置のようなオブジェクトへのアクセスは位置に独立 (location-independent) の方法でなされる。このようにして、オブジェクトアクセスレベルの透過性によって、任意のノードにおけるプログラムの自由な起動が可能となる。もちろん、オブジェクトアクセスレベルでの透過性をサポートするために、システムは透過的なオブジェクトの名前及び位置に関する機構を提供しなければならない。

2. **システム呼び出し及びプロセス間通信レベル**。移動の後に、移動させられたプロセスが元のノードに依存し続けないように、プロセス間通信を含めて、すべての**システム呼び出し (system call)** は位置独立である。このようにして、このレベルでの透過性は、先取り権付きのプロセス移動機能をサポートするシステムにおいて提供されねばならない。しかし、あるノードの物理的資源を要求するシステム呼び出しは位置独立である必要はない。

プロセス間通信の透過性は、新しく移動されたプロセスの過渡状態の間、メッセージの透過的宛先変更のためにも望ましい。すなわち、一度メッセージが送信されると、メッセージが受信される前に受信プロセスが別のノードに移動する場合に、送信ノードから再送することなく受信プロセスに到達すべきである。

最小干渉

プロセスの移動による、関連するプロセスの進行及び全体としてのシステムに引き起こす干渉は最小であるべきである。これを達成する 1 つの方法は、プロセスが移動されている間の凍結期間を最小にすることである。**凍結期間 (freezing time)** は、プロセスがその情報を宛先ノードへ転送している間プロセスが停止させられている時間として定義される。

最小残留依存性

残留依存性 (residual dependency) は移動前のノードに残されてはならない。すなわち、移動されたプロセスは、新しいノードで一度実行を開始すると、どのような形であれ、前のノードに依存し続けてはならない。そうでなければ、次のことが起きる。

■ 移動されたプロセスは前のノードに負荷をかけ続ける、したがって、プロセスを移動させる意味がない。
■ 前のノードの故障又は再立ち上げがプロセスの実行を妨げる。

効率

効率はプロセス移動を実装する際のもう 1 つの主要な問題である。プロセス移動に含まれる主な非効率の要因には次のものがある。

■ プロセスを移動するのに要する時間
■ オブジェクト (移動されるプロセスを含めて) の位置を特定するためのコスト
■ プロセスが移動された後、遠隔での実行に要するコスト

これらのコストのすべては実現できる範囲で最小に維持されなければならない。

堅固性

プロセス移動機構は、堅固 (robust) でなければならない。すなわち、プロセスが現在動作しているノード以外のノードが故障してもこのプロセスへのアクセス及び実行が妨げられてはならない。

ジョブのコプロセスとの間の通信

プロセス移動のもう 1 つの活用は、複数のノードに分散した単一のジョブのプロセス (すなわちコプロセス (coprocess)) 間での並列処理である。さらに、もしこの機能が通信コストを削減するためにサポートされているならば、これらのコプロセスはそれらの位置に関係なく互いに直接通信できることが必要である。

8.2.2 プロセス移動機構

プロセスの移動は、上記の良好なプロセス移動機構の要件を満たすために、多くの関連動作の扱いを含む複雑な動作である。プロセス移動に拘わる 4 つの主な関連動作には次のものが含まれる。

1. 送信元ノードにおけるプロセスを凍結し、宛先ノードで再開始する。
2. 送信元ノードから宛先ノードへの、プロセスのアドレス空間の転送
3. 移動プロセス (migrant process) 宛のメッセージのフォワーディング
4. プロセス移動の結果として分割されて、(異なるノードに配置された) 協力しているプロセス (コプロセス) 間での通信処理

これらの関連動作を扱う機構を以下に述べる。

プロセスの凍結及び再開機構

先取り権付きのプロセス移動において、通常の処理は、送信元ノードにおいてプロセスの状態の「スナップショット」をとり、宛先ノードでスナップショットを元に戻すことである。このために、移動中のある時点で、送信元ノードでプロセスを凍結し、その状態情報は宛先ノードに転送され、そしてプロセスはこの状態情報を用いて宛先ノードにおいて再開する。プロセスを凍結することによって、プロセスの実行は保留され、そのプロセスとのすべての外部とのやりとりは遅らされる。凍結及び再開の操作はシステムごとによって異なるが、

これらの操作に含まれる一般的事項について以下に述べる。

プロセスの即時及び遅延ブロック. プロセスが凍結される前に、実行がブロックされなければならない。プロセスの現在状態によって、即時にブロックされるか又は、プロセスがブロックできる状態になるまで遅延されなければならない。幾つかの典型的な場合を以下に述べる [Kingsbury 及び Kline 1989]。

1. もしプロセスがシステム呼び出しを実行していないならば、プロセスは即時にブロックできる。
2. もしプロセスがシステム呼び出しを実行しているが、カーネルの事象発生を待ちながら割り込み可能な優先度 (受信シグナルが何であってもそのプロセスを再開できる優先度) で休止しているならば、プロセスは即時にブロックできる。
3. もしプロセスがシステム呼び出しを実行し、カーネルの事象発生を待ちながら割り込み不可の優先度で休止しているなら、即時にブロックできない。プロセスのブロックはシステム呼び出しが完了するまで遅らされる。それゆえ、この場合、システム呼び出しが完了したことをプロセスに知らせるフラグがセットされ、自分自身をブロックする。

このブロック処理の一般的な手続きに、例外もあることに注意されたい。例えば、あるカーネルスレッドで実行中のプロセスは、割り込み不可の優先度で休止していても、即時にブロックできる。実際の機構は実装ごとに異なる。

高速及び低速 I/O 操作: 一般に、プロセスがブロックされた後、プロセスの凍結において次のステップはプロセスに付随するすべての高速 I/O(例えば、ディスク I/O) 操作の完了を待つことである。プロセスはすべての高速 I/O 操作の完了の後に凍結される。プロセス凍結の前に高速 I/O 操作が完了するのを待つことは実現性があることに注意されたい。しかし、パイプ (pipe) 処理や端末の操作のように低速 I/O 操作を待つことは現実的ではない。プロセスはプロセス移動の効率性のために時宜を得て凍結されなければならない。したがって、宛先ノードにおいてプロセスが実行を開始した後に、これらの低速 I/O 操作を正しく継続することが必要である。

オープンファイルの情報: プロセスの状態情報はプロセスが現在オープンし

ているファイルに関する情報を含んでいる。これらの情報はファイルの名前又は識別子、アクセスモード、及びファイルポインタの現在位置のような情報を含む。ネットワーク透過な実行環境を提供する分散システムにおいて、この状態情報を収集するのは問題ではない。その理由は、システム全体でのユニークなファイル識別子を使用して遠隔ファイルと同様にローカルファイルをアクセスするために、同じプロトコルが使用されるからである。しかし、幾つかの UNIX ベースのネットワークシステムは完全パス名でファイルを特定する [Mandelberg 及び Sunderam 1988, Alonso 及び Kyrimis 1988]。この場合、これらのシステムでは、UNIX ファイルシステムのセマンティクスの事情により、実行中のプロセスがファイルの完全パス名を獲得することは困難である。パス名は一度ファイルがオープンされるとその意味を失う。その理由は、オペレーティングシステムは、そのファイルへのすべての I/O 操作を行うためにプロセスが使用するファイル記述子をプロセスに戻すからである。それゆえ、このようなシステムでは、移動されたプロセスがそのファイルへのアクセスを継続できるように、何らかの方法でファイルへのポインタを保存する必要がある。次の 2 つの方法はこの目的のために使用される。

1. 最初の方法 [Mandelberg 及び Sunderam 1988] では、ファイルとパス名とのリンクが生成され、プロセスが移動された後はファイルへのアクセスポイントとして使用される。プロセス状態のスナップショットが作成されるとき、リンク (特別の名前をもつ) がそのプロセスによって使用されている各ファイルについて生成される。

2. 第二の方法 [Alonso 及び Kyrimis 1988] では、オープンしているファイルの完全パス名が要求時に再構成される。この実現のために、UNIX カーネルに必要な修正が取り入れられなければならない。例えば、[Alonso 及び Kyrimis 1988] において述べられた方法では、オープンされたファイルについての情報を格納する各ファイル構造が、そのファイルの完全パス名を含む動的に配置された文字列へのポインタによって補強される。

別のファイル問題は、元のノードのプロセスによって使用されていた 1 つ又はそれ以上のファイルが宛先ノードでも現れるということである。例えば、**nroff**、**cc** のようなシステムコマンドに対するプログラムコードは各ノードに複製される。これらのファイルを、前のノードからネットワークを通してアク

セスするよりも、プロセスが実行されているローカルノードからアクセスする方がより効率的であろう [Agrawal 及び Ezzat 1987]。別の例は、一時的ファイルはプロセスが実行されているノードに生成される方がより効率的であろう。これらのファイルは操作が終了すると、自動的に削除される。それゆえ、性能上の理由から、送信元ノードで収集されたファイル状態情報は、可能ならば遠隔ファイル操作の代わりにローカルファイル操作となるように、プロセスの転送宛先ノードで適切に修正されるべきである。これはまたアドレス空間転送時に転送されるデータの量を減少させる。宛先ノードに存在するファイルは転送の必要がないからである。

宛先ノードでのプロセスの復元: 宛先ノードで、プロセスが生成されるときの状態に類似した空きのプロセス状態が生成される。実装によっては、新しく配置されたプロセスは移動してきたプロセスと同じ識別子をもつ場合ともたない場合がある。ある実装では、この新しく生成されたプロセスのコピーは、古いコピーと新しいコピーの両方が存在でき、同時にアクセスできるために、最初は、移動してきたプロセスとは異なるプロセス識別子をもっている。しかし、もしプロセスの新しいコピーのプロセス識別子が古いものと異なるならば、宛先ノードでプロセスが実行を開始する前の段階で、新しいコピーの識別子は元の識別子に変更される。システムの他の部分では、両者の操作が保留されているので、プロセスの 2 つのコピーの存在を検出できない。移動プロセスのすべての状態が送信元から宛先ノードに転送され、空のプロセス状態にコピーされると、プロセスの新しいコピーが凍結解除され、古いコピーが削除される。このようにして、移動の前にプロセスがどのような状態であっても、宛先ノードでプロセスが再開する。

移動プロセスを復元する上記の方法は最も単純で直接的な場合である。特殊な場合には、特殊な処理と追加の処理が必要となろう。例えば、スナップショットを得ようとするとき、システム呼び出しのあるものは原子的ではないので、プロセスはシステム呼び出しを実行中かもしれない。特に、前に述べたように、低速の装置の I/O 操作を実行しているときにプロセスが凍結されると、この状況となる。もしスナップショットがこのような状況でとられると、システム呼び出しが再び実行されるときに限ってプロセスの継続が正しく行われる。それゆえ、通常これらの条件の検査が行われ、もし必要なら、システム呼び出しを再発行できるようにプログラムカウンタが調整される。

アドレス空間転送機構

プロセスは、プログラムのデータ、スタック、及び状態に従い実行されるプログラムからなっている。したがって、プロセスの移動は送信元ノードから宛先ノードへの、次の情報の転送を含んでいる。

■ プロセスの状態: これは実行状態 (レジスタの内容)、スケジューリング情報、プロセスによって使用されている主メモリに関する情報、I/O 状態 (I/O 待ち行列、I/O バッファの内容、割り込み信号、その他)、プロセスがアクセスする権利をもつオブジェクトのリスト (ケーパビリティリスト)、プロセスの識別子、プロセスのユーザとグループの識別子、プロセスによってオープンされているファイルについての情報 (モードや現在のファイルポインタの位置)、等を含んでいる。

■ プロセスのアドレス空間 (プログラムのコード、データ、及びスタック)

意味のあるプロセスにとって、プロセスのアドレス空間のサイズ (数メガバイト) はプロセスの状態情報 (数キロバイト) よりも格段に大きい。したがって、プロセス移動のコストはそのアドレス空間の転送に必要な時間が大部分である。移動プロセスの状態情報を転送する間は、その実行を完全に停止する必要があるが、プロセスのアドレス空間の転送は実行を停止しなくても可能である。さらには、移動プロセスの状態情報は、宛先ノードで実行を開始する前に、転送しておかなければならない。この反対に、プロセスのアドレス空間は、宛先ノードで実行を開始する前でも後でも転送することができる。

したがって、すべてのシステムにおいて、状態情報の転送の間は、移動プロセスの実行は停止させられる。しかし、プロセスのアドレス空間転送は移動決定の後であればいつでも良いとの柔軟性により、現存の分散システムでは、次の 3 つのアドレス空間転送機構のうちの 1 つを使用している。それらは、全体凍結、事前転送、又は参照時転送である。

全体凍結. この方法では、アドレス空間の転送の間は、プロセスの実行は停止される (図 8.2)。この方法は DEMOS/MP[Powell 及び Miller 1983]、Sprite [Douglis and Ousterhout 1987]、及び LOCUS [Popek and Walker 1985] によって使用され、実装が容易である。その主な欠点は、もしプロセスが移動の間に長期間保留されるなら、タイムアウトが発生し、ユーザによって遅延が感じられることである。

事前転送 (pretransferring):この方法では、プロセスがまだ送信元ノード

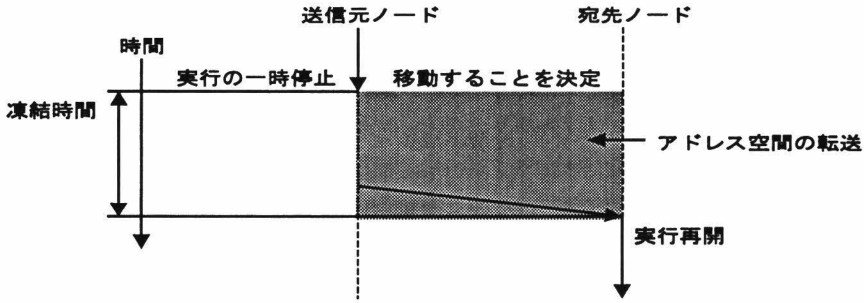


図 **8.2**　全体凍結機構

で動作している間にアドレス空間が転送される (図 8.3)。それゆえ、プロセスを移動するという決定がなされても、アドレス空間が宛先ノードに転送されるまで、送信元ノードで動作し続ける。事前転送 (**事前コピー (precopying)** としても知られる) は完全アドレス空間の初期転送としてなされ、それから前の転送との間に修正されたページの数が比較的小さくなるまで、又は修正されたページの数の著しい減少がなくなるまで、修正されたページ (ダーティビットを使用して検出) の転送が繰り返される。残りの修正されたページは状態情報の転送のために、プロセスが凍結された後に再転送される [Theimer 他 1985]。

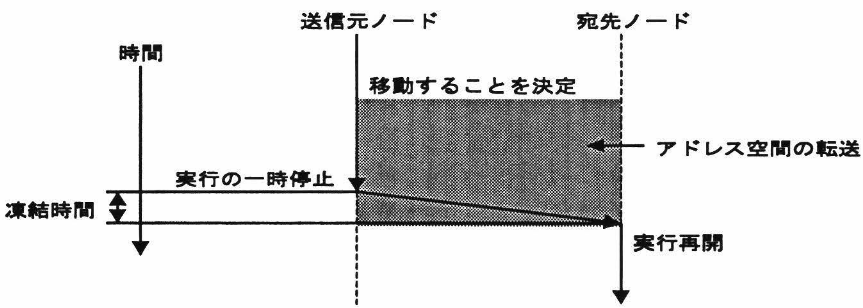


図 **8.3**　事前転送機構

事前転送操作において、最初の転送操作は全体のアドレス空間を移動させ、最長時間を要する、したがって、プログラムのアドレス空間に修正が起きるのことに対しても最長時間を与えている (その間に修正が起きやすいの意味)。第

2の転送は最初の転送の間に修正があったアドレス空間のページのみを転送する、したがって、時間が少なくなり、修正も少なくなり、だんだん転送すべきページが少なくなる、ついにはゼロに収束するか、ごく少数のページとなり、これらのページはプロセスが凍結された後に転送される。事前転送操作は、他のプログラムが事前転送操作の処理と干渉しないように、他のすべてのプログラムよりも高い優先度で送信元ノードで実行されることに注意されたい。

この方法は V-System [Theimer 他 1985] において使用された。この方法では、他のプロセスやユーザとプロセスとの相互動作に伴う移動の干渉が最小になり、凍結時間は減少される。事前転送はプロセスの凍結時間を減少させるが、**重複したページ (redundant page)** の転送が原因で、移動の全体時間が増加するであろう。重複したページは、事前転送が行われている間にダーティ(書き換えられた) となるので、1度より多く転送されるページである。

参照時転送 (transfer on reference): この方法は、プロセスは実行中はアドレス空間の比較的狭い範囲の部分のみを使用しがちであるという仮定に基づく。この方法では、プロセスのアドレス空間は送信元ノードに残されており、移動させられたプロセスは宛先ノードで実行するので、メモリのページを参照しようとすると、遠隔 (送信元) のノードから指定するブロックへのコピーするための要求を生成することになる。それゆえ、この要求時起動の参照時コピーの方法において、移動プロセスのアドレス空間のページが、送信元ノードから宛先ノードに参照時に限り転送される (図 8.4)。しかし、 Zayas [1987] はまた彼のシミュレーション結果によって、遠隔故障に対して、1つ隣接のページのプリフェッチは性能を改善することを示した。

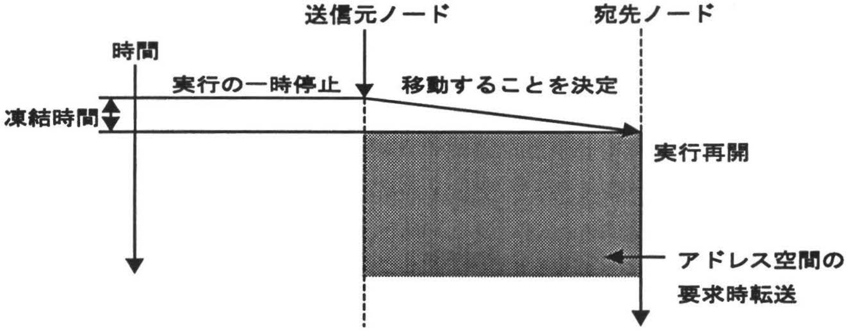


図 **8.4** 参照時転送機構

この方法は Accent [Zayas 1987] において使用された。この方法において、プロセスの移動が決定されると、送信元ノードから宛先ノードへのプロセスの切り替え時間は非常に短い、また実質的にアドレス空間のサイズに依存しない。しかし、この方法は、プロセスが移動された後の、遠隔実行をサポートするコストの点では効率的ではなく、転送を省略することによって節約された対価はプロセスがメモリを遠隔からアクセスするときに支払われなければならない。さらには、この方法はプロセスの送信元ノードでの負荷が継続し、送信元ノードの故障又は再起動があると、プロセスの故障という結果になる。

メッセージ転送機構

プロセスの移動に伴って、すべての保留 (pending)、移動中 (en-route)、及び将来 (future) のメッセージがプロセスの新しい位置に到着することが保証されなければならない。移動プロセスの新しい位置に転送されるべきメッセージは以下のように分類される。

タイプ 1: プロセスの実行停止後に、宛先ノードでプロセスの実行がまだ開始していないときに、送信元ノードで受信されるメッセージ

タイプ 2: 宛先ノードでプロセスの実行を開始した後に、送信元ノードで受信するメッセージ

タイプ 3: 宛先ノードでプロセスの実行を開始した後に、他のノードから移動プロセスに送信されるべきメッセージ

現存する分散システムにおいてメッセージ転送に使用されている機構を以下に述べる。

メッセージ再送機構: この機構は V-System [Cheriton 1988, Theimer 他 1985] 及び Amoeba [Mullender 他 1990] において、3 つのタイプのすべてのメッセージを扱うために使用された。この方法において、メッセージが配信不可能又は単純に欠落したとして、メッセージの送信者がデータのコピーを格納し再送の準備ができていることを前提として、タイプ 1 及びタイプ 2 のメッセージは送信者に返される。

例えば、V-System において、タイプ 1 又は 2 のメッセージは単に欠落となり、送信者はプロセスの新しいノードに再送することを促される。V-System のプロセス間通信機構は送信者は応答を受け取るのに成功するまで、再試行することになっている。同様に Amoeba において、タイプ 1 のすべてのメッセー

ジに対し、送信元ノードのカーネルは「後で再度試みよ、このプロセスは凍結されている」とのメッセージを送信者に送信する。送信元ノードからプロセスが削除された後に、これらのメッセージがいつかの時点で到着するが、その時はタイプ 2 メッセージである。すべてのタイプ 2 メッセージに対し、送信元ノードのカーネルは「このプロセスは当ノードにおいて未知である」というメッセージを送信者に送信する。

この方法において、否定応答を受信すると、送信者は、新しいプロセスの所在を見つけるために、**位置特定 (locate)** 操作を行う。V-System も Amoeba もプロセスの位置特定のために (オブジェクト位置特定機構を第 10 章に述べる) ブロードキャストメッセージを使用する。明らかに、この機構において、タイプ 3 のメッセージはプロセスの宛先ノードへ直接的に送信される。

この方法はプロセスの送信元ノードに、どのようなプロセス状態も残すことを要求していない。しかし、この機構の主な欠点は、プロセス移動操作のメッセージ転送機構は、移動プロセスとやり取りする他のプロセスにとって非透過的 (nontransparent) である。

発生元サイトの機構: この方法は AIX の TCF(透過的計算機能) において使用された [Walker 及び Mathews 1989] 及び Sprite [Doouglis 及び Ousterhout 1987]。これらのシステムのプロセス識別子は、その中に埋められたプロセスの**発生元サイト (origin site)**(又は**ホームノード (home node)**) をもっている、また各サイトは自己が生成した各プロセスの現在位置についての情報を維持する責任がある。それゆえ、プロセスの現在位置は、その発生元サイトに相談することにより、簡単に得られる。したがって、これらのシステムにおいて、特別なプロセスへのメッセージは常に発生元サイトに最初に送信される。発生元サイトは、それからメッセージをプロセスの現在位置に転送する。この方法は、発生元サイトの故障はメッセージ転送機構を中断させるので、信頼性の観点から良くない。この機構の別の欠点は、プロセスが発生元ノードから移動された後も移動プロセスの発生元サイトに負荷が連続することである。

リンク横断機構: DEMOS/MP [Powell 及び Miller 1983] において、タイプ 1 のメッセージを宛先変更するために、移動プロセスのための待ち行列が送信元ノードに生成される。このタイプのすべてのメッセージはこのメッセージ待ち行列に置かれる。プロセスが宛先ノードで立ち上がったと知らされると、待ち行列のすべてのメッセージは、移動手続きの一部として、宛先ノードに送信される。

タイプ 2 及び 3 のメッセージを宛先変更するために、**リンク**として知られる

転送アドレスが送信元ノードに残されている。リンクは移動プロセスの宛先ノードを指している。リンクの最も重要な部分はメッセージプロセスアドレスであり、2 つの要素をもっている。最初の要素は、システム全体においてユニークであるプロセス識別子である。これはプロセスが生成されたノードの識別子とそのノードによって生成されたローカルの識別子からなっている。第 2 の要素は、最後に認識されているプロセスの位置である。リンクの存在期間中、そのアドレスの第 1 要素は決して変わらないが、第 2 要素は変わるかもしれない。このようにして、タイプ 2 及び 3 のメッセージを転送するために、移動されたプロセスは一連のリンクを横断することによって位置が求められる。この一連のリンクはプロセスが発生したノードから始まって、プロセスの現在の位置までつながっている。リンクの第二の要素は、対応するプロセスがあるノードからアクセスされるとき更新される。これは、そのノードから同じプロセスに対する後続の位置特定操作の効率を改善するためになされる。

メッセージ転送のための DEMOS/MP によって使用されているリンク横断機構は効率と信頼性の面で欠点がある。あるノードからプロセスの位置を特定するために、幾つかのリンクを横断しなければならない。そしてもし一連のリンクのどれかのノードが故障していると、プロセスの位置を決めることができない。

リンク更新機構: Charlotte [Artsy 及び and Finkel 1989] において、プロセスは、位置独立のリンクを通して通信を行う。これらのリンクは 2 重の通信チャネルのための機能である。移動プロセスの転送フェーズの間、送信元ノードは、移動プロセスの通信相手のすべてを制御しているカーネルに、リンク更新メッセージを送信する。これらのリンク更新メッセージは、移動プロセスによって保持されている各リンクの新しいアドレスを通知し、同期する目的で通知されたカーネルによって確認される。このタスクは並列に行われるので、効率的である。この時点以降、リンク上のどこかにいる移動プロセス宛に送られるメッセージも、移動プロセスの新しいノードに直接送信される。移動プロセスが転送中の間に延期された通信要求は送信元ノードでバッファされ、移動手続きの一部として、宛先ノードに振り向けられる。それゆえ、タイプ 1 及び 2 のメッセージは、送信元ノードによって宛先ノードに転送される。そして、タイプ 3 のメッセージはプロセスの宛先ノードに直接送信される。

コプロセスを扱う機構

プロセス移動をサポートするシステムにおいて、もう 1 つの問題は、プロセ

ス (親 (parent)) と、それとは異なるノードに移動し配置されるかもしれないサブプロセス (子 (children)) との間の効率的な通信を提供することが必要である。この問題に関して、現存の分散オペレーティングシステムに使用されている 2 つの異なる機構を以下に述べる。

コプロセスの分離を認めない: コプロセス間の通信を扱う最も安易な方法は、それらのノード間の分離を認めないことである。これは次のようにして達成される。

1. 1 つ以上の子プロセスが完了するのを待っているプロセスの移動は認めない。
2. 親プロセスが移動するとき、子プロセスが必ず付随して移動されるようにする。

第一の方法は、幾つかの UNIX ベースのネットワークシステムによって使用されている [Alonso 及び Kyrimis 1988, Mandelberg 及び Sunderam 1988]。第二の方法は V-System [Theimer 他. 1985] によって使用されている。親プロセスが常に、子プロセスを伴い移動されることを確実にするために、V-System は**論理ホスト (logical hostl)** の概念を導入した。V-System のアドレス空間及び関連するプロセスは論理ホストにグループ化される。V-System のプロセス識別子は **(logical-host-id, local-index)(論理ホスト ID、ローカルインデックス)** の対として構成される。つまり、各プログラムは各自の論理ホストにおいて動作する。1 つのノードに複数の論理ホストが存在するかもしれないが、1 つの論理ホストは 1 つのノードに存在する。V-System において、あるプロセスのすべてのサブプロセスは 1 つの論理ホストの中で実行する。プロセスの移動は、実際にそのプロセスを含む論理ホストの移動である。したがって、典型的に、すべてのサブプロセスはプロセスが移動されるときに一緒に移動される。

この方法の主な欠点は、ジョブの中で並列処理を使用できないことである。ジョブの並列処理は、ジョブの中の様々なタスクを異なるノードに割り当て、これらのノードの上で同時に実行することにより行われる。さらには、V-System によって採用された方法において、論理ホストに幾つかの関連プロセスを含んでいるとき、プロセスを移動するのに伴うオーバヘッドは大きい。

ホームノード又は発生元サイトの概念. Sprite [Douglis 及び Ousterhout 1987] は、プロセスとサブプロセスとの間の通信に、この 2 つが別のノードで動作しているとき、**ホームノード**の概念 (前に述べた) を使用している。V-System

とは異なり、プロセス及びそのサブプロセスが独立に移動し、異なるノードで実行することが完全に自由である。しかし、親プロセスと子プロセスとの間のすべての通信はホームノード経由で行われるので、メッセージトラフィック及び通信コストは少なからず増大する。同様の欠点が LOCUS [Popek 及び Walker 1985] の発生元サイトの概念についても伴う。

8.2.3　異種システムにおけるプロセス移動

同種環境においてプロセスが移動させられるとき、送信元ノード及び宛先ノードの両方でデータの解釈は一貫している。それゆえ、**データ変換 (data translation)** の問題は生じない。しかし、異種環境 (heterogeneous environment) でプロセスが移動されると、宛先ノードで実行される前に、送信元 CPU フォーマットから宛先 CPU フォーマットに、関係するすべてのデータは変換されなければならない。もしシステムが 2 つの CPU タイプからなっていると、各プロセッサは、他のプロセッサタイプから自己のタイプのフォーマットにデータを変換できなければならない。もし第 3 の CPU が加えられると、各プロセッサは自己の表現と他の 2 つの表現との間を変換できなければならない。したがって、一般に、n 種の CPU タイプを持つ異種システムは、任意のノードから他のノードへのプロセスの移動機能をサポートするために、$n(n-1)$ 種の変換ソフトウェアを持たなければならない。4 つのプロセッサタイプの例を図 8.5(a) に示す。新しい CPU タイプを追加すると一層複雑になるので、この状況は望ましくない。

Maguire 及び Smith [1988] は、この変換処理の複雑性を減少するために、**外部データ表現機構 (external data expresenation mechanism)** の使用を提案した。この機構において、ある標準表現がデータの移送のために使用される。そして各プロセッサは標準形式との間での変換の能力を必要とするのみである。これが変換ソフトウェアの複雑性を制限した。4 つのプロセッサの例を図 8.5(b) に示す。特定のマシン表現から外部データ表現フォーマットに変換する処理は、**シリアライジング (serializing)** と呼ばれ、その逆の処理は**デシリアライジング (deserializing)** と呼ばれる。

標準データ表現フォーマットは**外部データ表現 (external data representation)** と呼ばれる。外部データ表現の設計者は、文字、整数、及び浮動小数点のように異なるデータ表現の問題を首尾よく扱わなければならない。中でも、浮動小数点の取り扱いは特別の注意が必要である。浮動小数点を外部デー

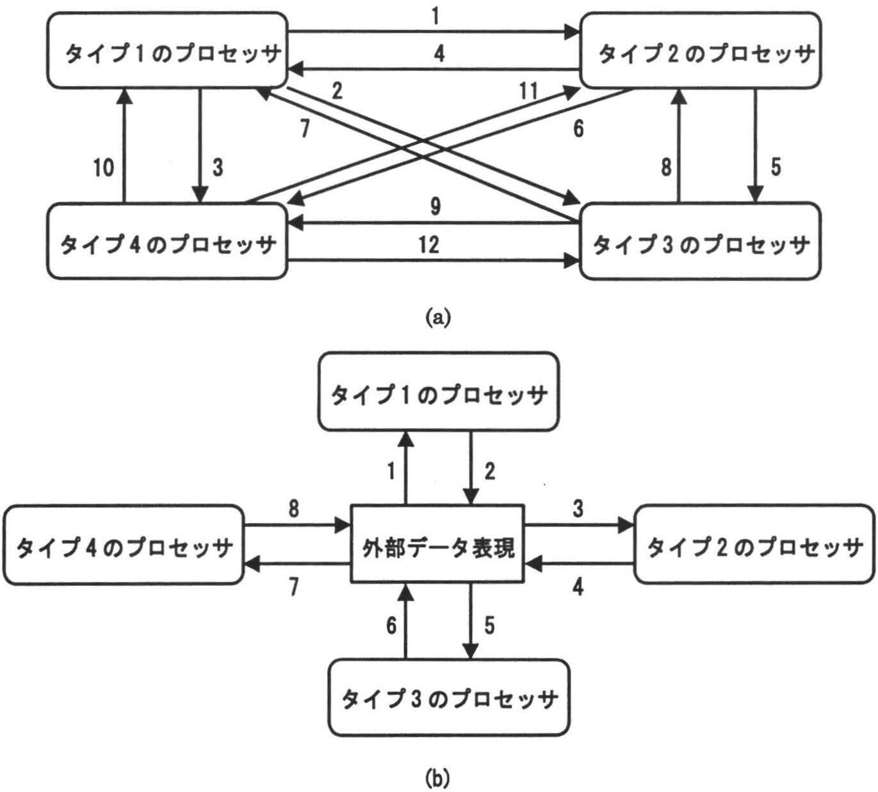


図 8.5 (a)4 つのタイプのプロセッサをもつ異種システムに要求される変換ソフトウェアは 12 通り必要であることを示す例。(b)4 つのタイプのプロセッサをもつ異種システムで外部データ表現を用いると 8 通りのソフトウェアのみが必要であることを示す例。

タ表現方式で扱うための Maguire 及び Smith [1988] によって議論された問題を以下に述べる。

浮動小数点表現は指数部分 (exponent part)、仮数部分 (mantissa part)、及び符号部分 (sign part) からなっている。指数と仮数の取り扱いは別々に議論されてきた。その理由は外部データ表現によって引き起こされる副次効果は 2 つの要素のそれぞれに異なって影響を与える。

指数の取り扱い

浮動小数点の指数に使用されるビット数はプロセッサごとに異なる。指数に、プロセッサ A が 8 ビットを使用し、プロセッサ B が 16 ビットを使用し、そしてプロセッサ方式 A のユーザによって設計された外部データ表現は 12 ビット (余分の 4 ビットは安全のため) を準備しているとしよう。また、仮数として 3 つの表現は同じ数のビットを使用していると仮定する。

この状況において、浮動少数点の数の表現において問題なく、プロセスはプロセッサ A から B へ移動されることができる。2 ステップの指数の変換処理は 8 ビットから 12 ビットへ、及び 12 ビットから 16 ビットへの変換であり、両ステップともに変換されるデータに対して余裕のビットをもっているからである。しかし、指数が 12 より多いビットの指数である浮動小数点をもつプロセスは、プロセッサ B から A へ移動されることができない。この浮動小数点データは、指数表現に 12 ビットのみをもつ外部データ表現では表現できない。ここでの問題は外部データ表現の問題であり、指数として同じ 16 ビットを使用している 2 つのプロセッサの間でさえデータ転送ができない。その理由は外部データ表現はこの目的に 12 ビットのみ使用しているからである。この問題は、外部データ表現が、分散システムにおけるどのプロセッサの最大長の指数とも同じ指数ビットをもつことを保証することにより、解決できる。

第 2 のタイプの問題は、指数が 12 ビット以下で 8 ビットより多い浮動小数点数がプロセッサ B から A に転送されるときに生じる。この場合には、外部データ表現はこのデータを扱うのに十分なビットをもっているが、プロセッサ A はもっていない。したがって、この場合に、プロセッサ A は、外部データ表現からの変換についてオーバフロー又はアンダーフロー (指数の符号に依存して) とするか、又は結果が無意味であるとする。この問題に対し、3 つの可能な解決策がある。

1. 移動するプログラムによって使用される浮動小数点の数が、システム内の

プロセッサの最小の指数値 (訳注、ビット数) よりも小さい指数値をもつことを保証する

2. 大きいプロセッサの指数値をエミュレートする

3. プロセスの移動を、指数表現が少なくとも送信元ノードのプロセッサの指数表現と同等の大きさをもつノードにのみに移動を制限する

最初の解決法は浮動小数点の使用に重大な制限を課するし、厳密な科学的計算にとって受け入れ難い。第2の解決法は計算時間の面で実用的でない位に非効率である。それゆえ、移動の宛先を制限する第3の解決法は実現可能な方法に見える。このために、システムの各ノードは、このノードからプロセッサを移動するための宛先として使用できるノードのリストを維持している。

仮数の扱い

仮数を扱う最初の問題は、指数を扱う問題と同じである。指数フィールドがすべてのプロセッサで同じサイズであり、仮数表現にプロセッサAが32ビットを使用し、プロセッサBが64ビットを使用し、外部データ表現が48ビットを使用していると仮定する。指数の扱いで述べたのと類似する理由のために、この場合もまたプロセッサAからBへのプロセスの移動は問題がないであろう。しかしプロセッサBからAへのプロセスの移動は、計算が「半精度(half-precision)」となるであろう。これは計算結果の精度が重要な場合に受け入れ難い。指数を扱うのと同様に、この問題を解決するために、外部データ表現は最大の仮数を扱うのに十分な精度をもたなければならない。そして移動の宛先は送信元ノードと少なくとも同じ大きさの仮数を持つノードに制限すべきである。

仮数を扱う第2の問題は、一群のプロセッサ間を複数回移動することによる精度の喪失である。ここでの例のプロセッサA及びBは上記の一群のプロセッサのサブ集合である。この問題は仮数の場合に限られる、その理由は仮数における数ビットの喪失は計算精度をそこなうのみであるが、指数が1ビット以上喪失したならば、致命的である。複数回数の移動による精度の喪失は累積的であるように見える、したがって移動の連続は全体として計算を不正にしかねない。しかし、外部データ表現が、システムのどのプロセッサの最長の仮数を表現するのに十分なように適切に設計されているならば、結果としての精度は、システムのすべてのプロセッサにおける最小の精度 (仮数に対する最小のビット数) をもつプロセッサにおける計算精度よりも悪くはならない。これは

遠隔計算が最小精度をもつプロセッサの浮動小数点計算に関して、「過剰精度 (extra precision)」計算として見られるからである。

符号付き無限大、及び符号付きゼロ表現の扱い
他に 2 つの、外部データ表現の設計で考慮が必要な事項は、**符号付き無限大 (signed-infinity)** と**符号付きゼロ (signed-zero)** の表現である。符号付き無限大は、計算結果が大きすぎる (オーバフロー) か又は小さすぎる (アンダーフロー) ことを示す機能をもつアーキテクチャによってサポートされた値である。別のアーキテクチャは符号ビットのみ値をもっていて、他のビットがゼロを使用するかもしれない、これが符号付きゼロである。

この場合には、これらの表現がすべてのシステムではサポートされていないというのが問題である。それゆえ、変換アルゴリズムの設計で、このような状況をどのように扱うかについて、適切な決定がなされなければならない。しかし、設計において、あるプロセッサがこのような例外情報を利用できるように、又は廃棄できるように、外部データ表現はこれらの値に注意しなければならない。

8.2.4　プロセス移動の利点

プロセス移動機能は、以下の利点をユーザに提供するために、分散システムにおいて実装される。

1. **プロセスの平均応答時間削減:** あるノードのプロセスの平均応答時間は、そのノードの負荷が増大するにしたがって急激に増大する。プロセス移動機能は、重負荷ノードのプロセスの一部を空きプロセッサ又は処理能力を十分に使用していないプロセッサに移動し、そこで処理することにより、このようなプロセスの平均応答時間を削減するのに使用されるかもしれない。

2. **個別のジョブの高速化:** プロセス移動機能は 2 つの方法で、個別のジョブの高速化に使用されるかもしれない。最初の方法は、ジョブのタスクをシステム内の異なるノードに移動させて、複数タスクを並行に動作させることである。第 2 の方法はジョブを高速 CPU をもつノード又は何等かの理由 (例えば、ジョブの特定の資源要件のため) により最小のターンアラウンド時間をもつノードに移動させることである。もちろん、実行時間での利得は移動コストよりも大きくなければならない。

3. スループットの向上: プロセス移動をサポートしないシステムにおいては、すべてのノードのCPUは十分に使用されていない可能性が大きい。しかし、プロセス移動機能をもつシステムでは、すべてのノードのCPU能力は、適切な負荷バランス方策を使用することにより、より十分に使用される。これがシステムのスループットを改善する手段となる。さらには、プロセス移動機能は、システムのスループットを増加させるために、全体として、I/O及びCPU処理の混在した処理にも使用されるかもしれない。

4. 資源の効率的使用: 分散システムにおいて、異なるノードのCPU、プリンタ、格納装置、その他は異なる。それゆえ、プロセスの特性に依存して、最も効率的な形でシステム資源を使用するように、最も適切なノードに移動させることができる。これは、ハードウェア資源だけではなく、データベース、ファイルその他のソフトウェア資源についても、同じである。さらには、特定用途ハードウェア装置のように、プロセスによって遠隔アクセスができない資源もある。例えば、高速フーリエ変換又は配列処理を行うのに遠隔アクセスを提供するのは困難であるかもしれないし、又はこのアクセスは実時間の目的に使用できない位に遅いかもしれない [Smith 1988]。プロセス移動は、どのノードのプロセスからもこのような資源の使用を可能とする。プロセスをその資源の存在位置に移動して実行することができるからである。

5. ネットワークトラフィックの削減: あるプロセスが最も頻繁に使用する資源 (ファイル、プリンタ他) の近くにプロセスを移動することは、もしよく使用する資源をアクセスするコストの減少が、余り使用しない資源をアクセスするコストの増大を相殺するならば、システムのトラフィックを減少させることになる。一般的に、プロセスがデータリダクション (データを解析し、その量を減少させ、結果を生成する処理) をプロセスのサイズよりも大きいデータに対して行うときは常に、データの存在位置にプロセスを移動する方が有利である [Smith 1988]。プロセス移動によってネットワークトラフィックを減少させるもう1つの方法は、互いに頻繁に通信する2つ以上のプロセスをまとめて、システムの同じノードに移動させることである。

6. システム信頼性の改善: プロセス移動機能は、幾つかの方法でシステム信頼性を改善するのに使用されるかもしれない。1つの方法は、単純に致命的プロセス (critical process) を、システム内の他のノードよりも信頼性の高いノードに移動することである。もう1つの方法は致命的プロセスのコピーを他のノードに移動させて、元のプロセスとコピーのプロセスを並行に実行することである。最後に、ノードの漸次的機能縮退を自ら明白にする手動停止のような

故障モードにおいて、プロセスの連続的な実行のために、ノードのプロセスは、ノードが完全に停止する前に、別のノードに移動させることができる。

7. **システムセキュリティの改善:** 重要なプロセスは安全なノードに移動され実行される。安全なノードは一般的ユーザから直接のアクセスはできないので、そのプロセスのセキュリティは改善される。

8.3　スレッド

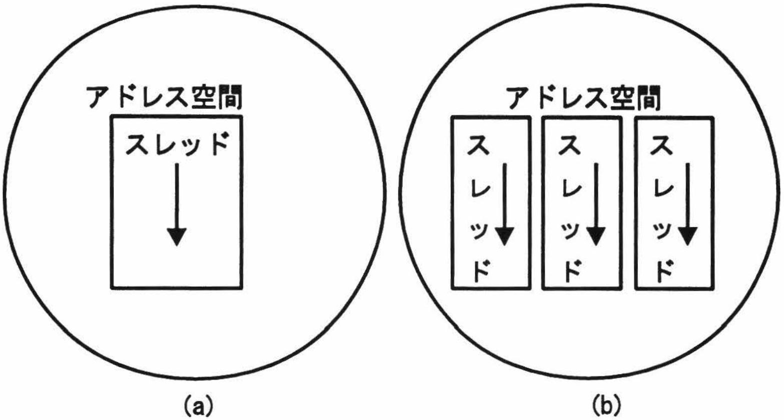


図 8.6　(a) 単一スレッドのプロセス、(b) 多重スレッドのプロセス。単一スレッドプロセスは従来のオペレーティングシステムのプロセスに対応する。

スレッド **(thread)** は並列処理 (parallelism) を用いてアプリケーション性能を改善する一般的な方法である。伝統的なオペレーティングシステムでは、CPU 使用の基本的単位は**プロセス (process)** である。各プロセスは各自のプログラムカウンタ、各自のレジスタセット、各自のスタック、及び各自のアドレス空間をもっている。他方、スレッド機能をもつオペレーティングシステムでは、CPU 使用の基本的単位はスレッドである。これらのオペレーティングシステムでは、1 つのプロセスは 1 つのアドレス空間と 1 つ以上の**制御のスレッド (thread of control)** からなっている (図 8.6(b))。プロセスの各スレッドは各自のプログラムカウンタ、各自のレジスタセット、及び各自のスタックをもっている。しかし、プロセスのすべてのスレッドは同じアドレス空間を共有する。したがって、スレッドは、オープンされたファイル、子プロセス、セマ

フォ、シグナル、課金情報、その他のようなオペレーティングシステム資源の同じ集合も共有する。アドレス空間の共有によって、プロセスのスレッド間に保護はない。しかし、これは問題ではない。異なるプロセスは異なるユーザに属していることもあるので、プロセス間の保護は必要である。しかし、1つのプロセス(及びそのすべてのスレッド)は常に一人のユーザによって所有されている。それゆえ、1つのプロセスの複数のスレッドの間では保護は不要である。もし1つのプロセスの2つのスレッド間の保護が要求されるならば、それらを1つのプロセスに配置させるよりも、それらを異なるプロセスに配置するのが適切である。

スレッドはプロセスと同じ方法でCPUを共有する。すなわち、ユニプロセッサ(単一プロセッサ)では、スレッドは**擬似並列(quasi-parallel)(タイムシェアリング)**で動作する。共有メモリマルチプロセッサでは、プロセッサの数だけ、スレッドは同時に動作する。さらには、従来のプロセスのように、スレッドは子スレッドを生成できるし、完了するためにシステム呼び出しを待つこともできるし、実行の途中で状態を変えることもできる。特定の瞬間において、スレッドは、動作中、ブロック中、待機中、終了の状態のどれかの状態にある。これらの類似性のゆえに、スレッドはミニプロセスとしばしばみなされる。実際に、スレッド機能をもつオペレーティングシステムにおいて、1つのスレッドをもつプロセスは従来のオペレーティングシステムにおけるプロセスに対応する(図8.6(a))。スレッドはしばしば**ライトウエイトプロセス(lightweight process)**と呼ばれ、従来のプロセスは**ヘビーウエイトプロセス(heavyweight process)**と呼ばれる。

8.3.1 スレッド使用の動機

ある計算処理を行うために、単一スレッドの複数プロセスの代わりに、複数スレッドのプロセスを使用する主な動機を以下に述べる。

1. 新しいプロセスを生成するオーバヘッドは、プロセス内にスレッドを生成するオーバヘッドよりも一般にかなり大きい。

2. 同じアドレス空間を共有するスレッド間の切り替えは、個別のアドレス空間をもつプロセス間の切り替えに比べてかなりオーバヘッドが小さい。

3. スレッドにより、逐次実行(sequential execution)と**ブロッキングシステム呼び出し(blocking system calls)**を組み合わせた処理の並列処理が可能

である [Tanenbaum 1995]。並列処理は性能を改善し、ブロッキングシステム呼び出しによりプログラム作成を容易にすることできる。

4. あるプロセスのスレッドは同じアドレス空間を共有するので、スレッド間の資源共有がプロセス間より効率的かつ自然にできる。

これらの利点について以下に説明する。

新しいプロセスの生成とその実行環境の設定のオーバヘッドは、存在しているプロセス内に新しいスレッドを生成するのに比べ、かなり大きくなりがちである。その理由は主に、新しいプロセスを生成するとき、アドレス空間の一部は親プロセスから継承するが、アドレス空間は最初から生成されなければならない。しかし、新しいスレッドを生成するときには、そのプロセスのアドレス空間を使用するので、最初から生成する必要はない。例えば、カーネルによってサポートされた仮想メモリシステムの場合、新しく生成されたプロセスは、データ及び命令が最初に参照されるとき、ページ異常を招くであろう。さらには、ハードウェアキャッシュは新しいプロセスに対し、最初はデータをもっていない。プロセスのデータに対するキャッシュエントリはプロセスの実行時に生成される。これらのオーバヘッドはスレッド生成でも発生するが、より小さいことが多い。その理由は、新しく生成されたスレッドが、そのプロセス内の他のスレッドによって最近アクセスされた命令とデータをアクセスするとき、スレッドは、ハードウェアやメモリキャッシュがどのようであれ、自動的にその利点を使用するからである。

スレッドはまたコンテキスト切り替え時間を最小にする。すなわち CPU が 1 つの計算単位から別の計算単位に切り替わるのを最小のオーバヘッドで行う。あるプロセスのスレッド間で、アドレス空間及びその他のオペレーティングシステム資源を共有することにより、スレッド間の CPU 切り替えのオーバヘッドは、各自のアドレス空間をもつプロセス間の CPU 切り替えに比べて、非常に小さい。これがスレッドがライトウエイトプロセスと呼ばれる理由である。

スレッドがどのように逐次実行とブロッキングシステム呼び出しのを組み合わせて並列処理を可能としているかを明白にするために、サーバプロセスが構成される異なる方法を考える。次の 3 つのモデルの 1 つがサーバプロセスを構成するのに使用される (例えば、ファイルサーバの場合を考える)[Tanenbaum 1995]。

1. **単一スレッドプロセス:** このモデルはブロッキングシステム呼び出しを使

用するが、並列処理は行わない。この方法において、ファイルサーバはクライアントのファイルアクセス要求を要求待ち行列から取り出し、アクセス許可を検査し、そしてもしアクセスが許可されると、要求を処理するのにディスクアクセスが必要か検査する。もし、ディスクアクセスが不要であれば、その要求は直ちに処理され、応答がクライアントプロセスに送られる。そうでなければ、ファイルサーバはディスクアクセス要求をディスクサーバに送り、応答を待つ。ディスクサーバの応答を受け取ると、クライアントの要求を処理し、クライアントプロセスに応答を送り、そして要求待ち行列から次の要求を取り出す。

この方法において、ブロッキングシステム呼び出しの使用によりサーバプロセスのプログラム作成は単純である。すなわち、要求を送った後、ファイルサーバは、応答がディスクサーバから受け取られるまでブロックする。しかし、もしファイルサーバとして専用マシンが使用されると、ファイルサーバがディスクサーバからの応答を待っている間、CPU は空きのままである。したがって、この方法では並列処理は行われず、単位時間あたりに処理されるクライアントからの要求の数はわずかである。

単一スレッドプロセスとして実装されたサーバの性能は時には受け入れ難いほど悪い。それゆえ、ファイルサーバが複数の要求を同時に処理できるようにして、複数のクライアントの要求を多重に処理する必要がある。次の 2 つのモデルは、この目的で並列処理をサポートするモデルである。

2. **有限状態機械モデル**: このモデルは**ブロッキングシステム呼び出しなし (nonblocking system calls)** で並列処理をサポートする。この方法では、サーバは単一スレッドプロセスとして実装され、有限状態機械のように運用される。クライアント要求メッセージ及びディスクサーバからの応答メッセージが待っている事象待ち行列が維持されている。スレッドが空きになると常に、事象待ち行列から次のメッセージを取り出す。もしそれがクライアント要求メッセージならば、アクセス許可とディスクアクセスの必要性が検査される。もしその要求を処理するためにディスクアクセスが必要ならば、ファイルサーバはディスクサーバにディスクアクセス要求を送信する。しかし、この方法では、ブロックする代わりに、クライアント要求の現在状態をテーブルに記録し、それから事象待ち行列から次のメッセージを取りだしに行く。このメッセージは新しいクライアントからの要求であるか、以前のディスクアクセス要求のディスクサーバからの応答であるかもしれない。もし、それが新しいクライアント要求であるなら、上記の処理がなされる。他方、もしそれがディスクサーバからの応答であるならば、応答に対応するクライアントの要求の状態がテー

ブルから更新される。そして、クライアントのサーバ要求がさらに処理される。

この方法は並列処理を実現するが、この方法でサーバプロセスをプログラムすることは、ブロッキングシステム呼びだしを利用していないので、困難である。サーバプロセスは、未完了の各クライアント要求のエントリ (要求されたこと) を保持しなければならない。そしてディスク操作が完了するたびに、その要求を実行し続ける方法を見つけるために、クライアントの該当する一部を更新しなければならない。

3. スレッドのグループ:このモデルはブロッキングシステム呼び出しを利用することにより並列処理をサポートしている。この方法では、サーバプロセスは、**1 つのディスパッチャスレッド**と複数の**ワーカ (worker)** スレッドからなっている。要求が入ってくると、ワーカスレッドは動的に生成されるか、又はスレッドの数と同じだけ、同時の要求を扱うために、開始時にスレッドのプールが生成される。ディスパッチャスレッドはクライアントからの要求をループして待ち続ける。クライアントの要求が到着すると、アクセス許可の検査がなされる。もし、許可が認められると、新しいワーカスレッドを生成するか、又はプールから空きのワーカスレッドを選び (ワーカスレッドが動的に生成されるのか、静的に生成されるかによって)、要求をワーカスレッドに引き渡す。制御はワーカスレッドに渡され、ディスパッチャスレッドの状態は動作中から待機中に変化する。ワーカスレッドは要求を処理するのにディスクアクセスが必要かどうか、又はすべてのスレッドによって共有されるブロックキャッシュから満たされるかを検査する。もしディスクアクセスが必要ならば、ディスクサーバにディスクアクセス要求を送信し、ディスクサーバからの応答を待つためにブロックする。この時点で、スケジューラが起動され、待機状態にあるスレッドのグループから実行するスレッドが選択される。選択されたスレッドはディスパッチャスレッドかもしれないし、又は待機中の別のワーカスレッドかもしれない。

この方法は、ブロッキングシステム呼び出しを行う逐次処理の考えを保持しながら、並列処理を実現する。それゆえ、この方法により設計されたサーバプロセスは性能が良く、プログラムすることも容易である。

サーバプログラムの設計において、スレッドを使用する動機について概観した。クライアントプロセスもスレッドによって実現される並行性から利益を受けることが多い。例えば、クライアントプロセスは、データを個別に処理可能な複数のブロックに分割するために、分割統治アルゴリズム (divide-and-conquer algorithm) を使用するかもしれない。そしてクライアントプロセスは各ブロッ

クごとに1つのサーバに送信し、最後にその結果を収集し組み合わせる。この場合に、個別のクライアントスレッドは各サーバと相互動作して使用される。あるクライアントのアプリケーションのユーザインタフェースはスレッドを使用することにより、長時間の操作が行われている間、インタフェースをユーザに戻すことができるといった利益を受ける。多くの分散操作を行うクライアントプロセスもまた、各操作をモニタする個別のスレッドを使用することにより、スレッドから利益を受ける。

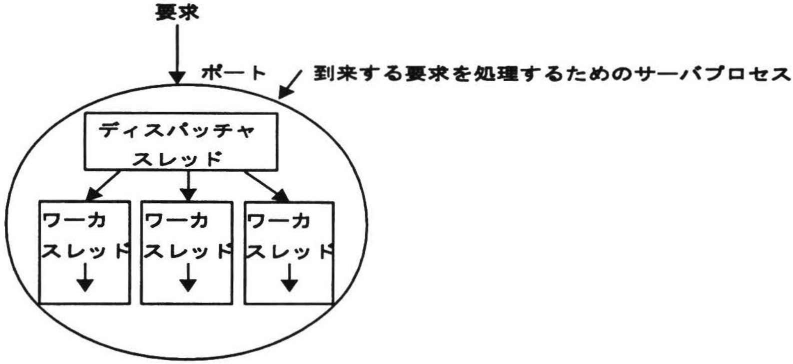
最後に、スレッドの使用は、共有アドレス空間を使用する一群のスレッドは多くのアプリケーションにとって最も自然な方法である。例えば、生産者-消費者モデルを使用するアプリケーションにおいて、生産者及び消費者プロセスは共通のバッファを共有しなければならない。それゆえ、生産者及び消費者が同じプロセスの2つのスレッドとなるようにアプリケーションプログラムを作成することにより、ソフトウェアをより単純にすることができる。

8.3.2 スレッドの組織化モデル

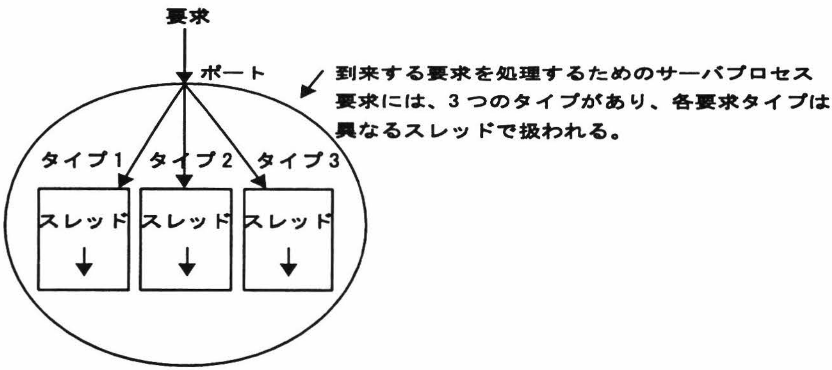
アプリケーションのニーズに応じて、アプリケーションプロセスのスレッドは異なる方法で組織化される。プロセスのスレッドを組織化するために通常使用される3つの方法を以下に述べる。

1. **ディスパッチャワーカモデル**: このモデルの使用をサーバプロセスの設計において、既に述べた。このモデルでは、プロセスは1つのディスパッチャスレッドと複数のワーカスレッドからなる。ディスパッチャスレッドはクライアントからの要求を受け付け、要求を調べた後に、要求のそれ以後の処理をワーカスレッドの一つに配分(ディスパッチ)する。各ワーカスレッドは異なるクライアント要求に対応し動作する。それゆえ、複数のクライアント要求は並列に処理される。このモデルの一例を図8.7(a)に示す。

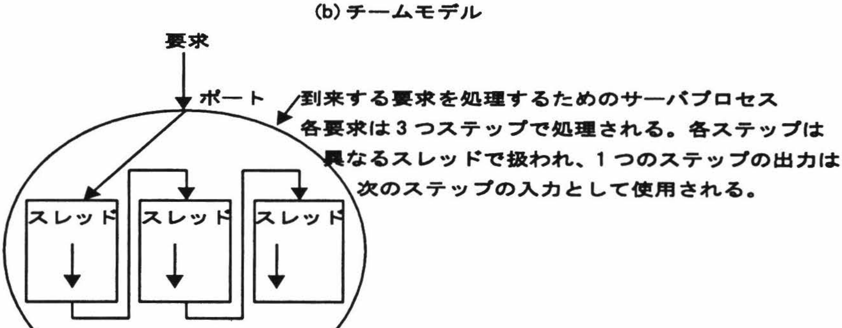
2. **チームモデル**: このモデルでは、すべてのスレッドは、クライアントの要求に対してディスパッチャンワーカといった関係が存在しないという意味で、平等に振る舞う。各スレッドは、各自自身で要求を獲得し、処理する。このモデルはプロセス内に特別スレッドを実装するためによく使用される。すなわち、プロセスの各スレッドは、特別のタイプの要求を処理するように特化されている。それゆえ、複数のタイプの要求はプロセスによって同時に処理される。このモデルの一例を図8.7(b)に示す。



(a) ディスパッチャンワーカモデル



(b) チームモデル



(c) パイプラインモデル

図 **8.7**　スレッドの組織化モデル

3. パイプラインモデル: このモデルは、アプリケーションの1つの部分によって生成された出力データは、アプリケーションの別の部分の入力になる生産者-消費者モデルに基づくアプリケーションに有用である。このモデルでは、第1のスレッドによって生成された出力データは第2のスレッドによって使用される、第2のスレッドの出力データは第3のスレッドによって使用され、以下同様なように、あるプロセスのスレッドはパイプラインとして組織化されている。パイプラインの最後のスレッドの出力はスレッドが所属するプロセスの最終的出力である。このモデルの一例を図 8.7(c) に示す。

8.3.3 スレッドパッケージの設計事項

スレッド機能をサポートするシステムは、スレッド関連操作のためにユーザに一式のプリミティブを提供しなければならない。システムのこれらのプリミティブはスレッドパッケージを形成すると言われる。スレッドパッケージを設計する際の重要な事項を以下に述べる。

スレッド生成

スレッドの生成は静的に又は動的になされる。静的な方法において、あるプロセスに属するスレッドの数はプロセスの存在期間にわたり固定であるが、動的な方法では、スレッドの数は動的に変化し続ける。動的な方法において、プロセスは1つのスレッドで始まり、新しいスレッドがプロセスの実行開始時及び実行中に生成され、1つのスレッドは自己のジョブを終了すると、終了呼び出しを行い、自身で消滅させることもある。他方、静的方法では、1つのプロセスのスレッドの数はプログラムを書く時、又はコンパイルの時に決定される。静的方法において、各スレッドに固定のスタックが割り付けられるが、動的方法では、スレッド生成のシステム呼び出しへのパラメータとして、スレッドのスタックサイズが指定される。このシステム呼び出しの他のパラメータは通常、スケジュールの優先度、及びこのスレッドを動作させるために実行されるべき手続きを含んでいる。システム呼び出しは新しく生成されたスレッドの識別子を戻す。この識別子はこのスレッドに含まれる後続の呼び出しに使用される。

スレッドの終了

スレッドの終了は従来のプログラムの終了と同様な方法でなされる。すなわち、スレッドは、終了呼び出しを行うことにより、ジョブを完了したとき、自

分自身を消滅させるか、又はスレッド識別子をパラメータに指定した強制終了コマンドを使用することにより外部から強制終了させられる。多くの場合に、スレッドは終了しない。例えば、静的に生成されたスレッドを使用するプロセスにおいて、スレッドの数は存在期間にわたって固定であると、先に述べた。このようなプロセスにおいて、すべてのスレッドはプロセスが開始すると直ちに生成され、これらのスレッドはプロセスが終了するまで、強制終了させられることはない。

スレッド同期

スレッドの同期について述べる。1 つのプロセスのすべてのスレッドは共通のアドレス空間を共有するので、複数のスレッドが同じデータを同時にアクセスしようとすることを防止しなければならない。例えば、1 つのプロセスの 2 つのスレッドがプロセスの中の同じグローバル変数を 1 加算する必要があると想定する。これが安全に行われるために、各スレッドはある時間間隔の間、この変数を排他的にアクセスできることを保証しなければならない。スレッドがある共有の変数をアクセスするかもしれないプログラムの部分は**臨界領域 (critical region)** と呼ばれる。複数のスレッドが同時に同じデータをアクセスすることを防止するために、1 つのスレッドが臨界領域の中を実行しているときに、他のスレッドが同じデータをアクセスする臨界領域を実行しないことを保証すれば十分である。すなわち、同じデータがアクセスされる臨界領域の実行は、同時には**相互に排他的 (mutually exclusive)** でなければならない。スレッドパッケージにおいて相互排他に通常使用される 2 つの相互排他技法は mutex 変数及び条件変数である。

mutex 変数 (mutex variable) は、ロック中又はアンロック中の 2 つの状態のうちの常に 1 つである 2 進数のセマフォのように機能する。臨界領域の中を実行しようとするスレッドは、対応する mutex 変数に**ロック (lock)** 操作を行う。もし mutex 変数がアンロック状態にあるならば、そのロック操作は成功し、mutex 変数の状態がアンロック中からロック中に、1 つの原子動作として変化する。この後、スレッドは臨界領域の中で実行できる。しかし、もし mutex 変数が既にロック中ならば、ロック操作は、実装に依存して、次の方法の 1 つで処理される。

1. スレッドがブロックされ、mutex 変数を待つスレッドの待ち行列に入れられる。

2. 失敗を示す状態コードがスレッドに返される。この場合、スレッドは他のジョブを継続する柔軟性をもっている。しかし、臨界領域に入るためには、スレッドは成功するまで mutex 変数をロックすることを試み続けなければならない。

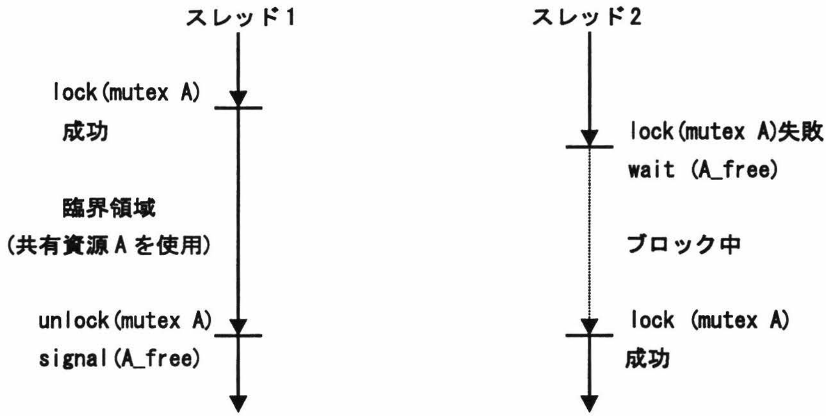
スレッドパッケージは、実際にロックを行う操作と mutex 変数の状態を獲得する操作の両方をサポートするかもしれない。

異なるスレッドが異なる CPU で並列に動作するマルチプロセッサシステムにおいて、2 つのスレッドが同時に同じ mutex 変数のロック操作を行うことが起きるかもしれない。そのような状況では、それらの 1 つが勝者となり、敗者はブロックさせられるか、失敗を示す状態コードを返される。

スレッドが臨界領域での実行を終えると、対応する mutex 変数に**アンロック (unlock)** 操作を行う。この時点で、もしブロック法が使用され、mutex 変数を待ってブロック中の 1 つ以上のスレッドがあるならば、それらの 1 つがロックを与えられ、その状態はブロック中から動作中に変化させられる。他のスレッドは待ち続ける。

mutex 変数は 2 つの状態をもつのみであるので、実装が単純である。しかし、その用途は臨界領域に入るのを保護するのに限られる。より一般的な同期要求に対して、条件変数が使用される。**条件変数 (condition variable)** は mutex 変数と関連し、その変数のブール代数状態を反映している。**ウエイト (wait)** 及び**シグナル (signal)** は条件変数に通常提供される 2 つの操作である。スレッドが条件変数にウエイト操作を行うとき、関連する mutex 変数はアンロック中であり、スレッドは、その条件変数に他のスレッドによってシグナル操作が行われるまで、ブロックされる。シグナル操作は待っていた事象が発生したかもしれないことを指示する。スレッドがシグナル操作を条件変数に行うとき、mutex 変数はロックされ、そして条件変数を待ってブロックされていたスレッドが臨界領域内で実行を開始する。図 8.8 は、スレッドの同期のために、mutex 変数と条件変数の使用を示す。

条件変数はしばしば、スレッド間の協調のために使用される。例えば、あるプロセスの 2 つのスレッドが生産者-消費者の関係にあるアプリケーションにおいて、生産者スレッドはデータを生成し、指定されたバッファに格納する。そして消費者スレッドはバッファからデータを取り出し、以後の処理のためにデータを使用する。この場合に、消費者スレッドが検査したとき、もしバッファが空であるならば、このスレッドは空でない条件変数を待たされる。そして生



mutex A は共有資源 A の排他的使用のための mutex 変数である。
A_free は資源 A がフリーになる(解放される)ための条件変数である。

図 8.8　スレッド同期のための mutex 変数及び条件変数の使用

産者スレッドがバッファにあるデータを格納するとき、空でない **(nonempty)** 条件変数をシグナルすることができる。同様に、生産者スレッドが検査するとき、もしバッファがフルならば、このスレッドはフルでない条件変数を待たされる。そして消費者スレッドがバッファからデータを取り出すとき、フルでない **(nonfull)** 条件変数をシグナルできる。この方法で、2 つのスレッドは、条件変数を使用することにより、互いに協調して動作することができる。

スレッドスケジューリング

スレッドパッケージの設計において別の重要な事項はスレッドをどのようにスケジュールするかということである。スケジュールパッケージは、アプリケーションで使用されるスケジューリング方策を指定する柔軟性をユーザに与えるための呼び出しを備えることが多い。この機能によると、アプリケーションのプログラマはスケジューリングのための最も有効な方法を決定する問題の発見的手法を使用することができる。スレッドパッケージによってサポートされているスレッドスケジューリングの特別な特性の幾つかを以下に述べる。

1. **優先度指定機能:** 単純なスケジューリングアルゴリズムでは、スケジュー

リングは先着優先方式によってスケジュールされるか又は、すべてのスレッドがスケジューリングアルゴリズムによって平等に取り扱われるように、最小使用時間ごとに、スレッドの間でCPUサイクルをタイムシェアするために、ラウンドロビン方策が使用される。しかし、スレッドスケジューリング方式では、アプリケーションプログラマに、重要な処理がより高い優先度で動作できることを保証するためにアプリケーションの各種のスレッドに優先度を指定する柔軟性を提供する。

優先度ベースのスレッドスケジューリング方式は先取り権なし又は先取り権付きのどちらかである。前者の場合、一度CPUがあるスレッドに割り付けられると、スレッドはそれがブロック、終了、又は配分時間使用し終わるまでCPUを使用できる。すなわち、たとえ別のより高い優先度のスレッドが**動作待機 (ready to run)** となっても、既に割り付けられたスレッドからCPUが取り上げられることはない。より高い優先度のスレッドは、現在CPUを使用しているスレッドが、解放した後にのみ動作できる。他方、先取り権付きの方式では、より高い優先度のスレッドは常に、より低い優先度のスレッドを先取りする。すなわち、より高い優先度のスレッドが動作待機となると、現在動作中のより低い優先度のスレッドが保留となり、CPUはより高い優先度のスレッドに割り付けられる。この方式では、他により高い優先度のスレッドが動作待機にないときに限り、あるスレッドは動作できる。

2. **配分時間サイズの動的変更の柔軟性:** 単純なラウンドロビンスケジューリング方式はスレッド間でCPUサイクルをタイムシェアするために、固定長の配分時間を割り付ける。しかし、固定長の配分時間はマルチプロセッサシステムにおいては適切ではない。その理由は、使用可能なプロセッサの数よりも動作可能なスレッドの数が少ないかもしれないからである。この場合、配分時間が過ぎて、スレッドに割り込みをかけてカーネルへコンテクスト切り替えを行っても、動作状態に直ちに戻るので、このような動作は浪費となるであろう。それゆえ、固定長の配分時間を使用する代わりに、スケジューリング方式は、システムにおけるスレッドの総数と反対に配分時間の長さを変化させるかもしれない。このアルゴリズムは、重負荷システムにおいては処理時間の短い要求に対して良好な応答時間となり、軽負荷システムでは高い効率を提供する。

3. **ハンドオフスケジューリング:** ハンドオフ (手渡し) スケジューリング方式では、スレッドが指定するとその後続者を指名することができる。例えば、別のスレッドにメッセージを送信した後に、送信スレッドはCPUを放棄し、受信スレッドが次に動作することを要求できる。それゆえ、この方式は動作可

能スレッドの待ち行列をバイパスし、現在動作中のスレッドから指定されるスレッドに直接CPUを切り替えるという柔軟性を提供する。ハンドオフスケジューリングは、上手に使用すれば性能を向上できる。

4. **関連スケジューリング**: マルチプロセッサシステムの性能改善に使用される別のスケジューリング方策は**関連スケジューリング (affinity scheduling)** である。この方式では、CPUのキャッシュにスレッドのアドレス空間の部分がまだ残っていることを期待して、そのスレッドが最後に動作したCPU上にスケジュールされる。

シグナルの取り扱い

シグナルはソフトウェアで生成された**割り込み**と**例外**を提供する。割り込みは、外部で生成されるスレッド又はプロセスの分断であり、例外はスレッドの実行中に通常でない条件の発生によって引き起こされる。マルチスレッド環境でのシグナルの扱いに関して2つの主な問題を以下に述べる。

1. シグナルは、プロセスのどのスレッドが受信しようとも、適切に処理されなければならない。UNIXにおいてシグナルのハンドラは、プロセスにおいてシグナルを受信するルーチンであることを思い出そう。

2. シグナルは紛失を防止しなければならない。シグナルの紛失は、最初のシグナルが発生したスレッドによって扱われる前に他のスレッドにおいて同じタイプの別のシグナルが発生するときに起きる。その理由は、シグナルを発生している例外条件は、プロセスにわたるグローバル変数に格納され、同じタイプのシグナルを発生する別の例外条件によって上書きされるからである。

前者の問題を扱うための方法は、各プロセスにおいて個別の例外ハンドラスレッドを生成することである。この方法では、プロセスの例外処理スレッドはプロセスのどのスレッドにおいても起きるすべての例外条件を処理する責任がある。スレッドが例外条件のシグナルを受信すると、スレッドは例外発生メッセージを例外ハンドラに送信し、例外が処理されるまで待つ。例外メッセージは通常、例外を発生した例外条件、スレッド、及びプロセスについての情報を含む。例外ハンドラスレッドは例外のタイプに応じてその機能をはたす。これは、例外の消去、犠牲スレッドの再開始、又は犠牲スレッドの終了を含む。

他方、後者の問題を扱うための方法は、スレッド間のグローバル変数の使用で競合がおきないように、各スレッドを例外条件をシグナルするためのそれ自

体の固有のグローバル変数に割り付けることである。スレッドのコード (プログラム) は通常複数の手続きからなるので、このような変数はスレッドにわたるグローバル変数と言われる。この方法において、これらのスレッドにわたるグローバル変数を生成し、設定し、及び読み取るために、新しいライブラリ手続きが必要である。

8.3.4 スレッドパッケージの実装

スレッドパッケージはユーザ空間又はカーネル空間のどちらかに実装される。以下の説明では、2つの方法をそれぞれ、ユーザレベル及びカーネルレベルと称する。ユーザレベルの方法では、ユーザ空間はスレッド管理ルーチンの集まりである実行時システムからなっている。スレッドはユーザ空間の実行時システムの直上で動作し、実行時システムによって管理される。実行時システムは、各スレッドの現在状態を追跡し続けるために、状態情報テーブルを維持する。このテーブルはスレッドごとに1つのエントリをもっている。このテーブルの1つのエントリは、スレッドのレジスタ値、状態、優先度、その他の情報をもっている。スレッドパッケージのすべての呼び出しは、呼び出しに対応した機能をはたす実行時システム手続きへの呼び出しとして実装されている。これらの手続きは、もし呼び出しを行ったスレッドが呼び出しの間、保留となるならば、スレッド切り替えも行う。すなわち、この方法では2レベルスケジューリングが行われる。カーネルの中のスケジューラは配分時間をヘビーウエイトプロセスに割り当て、実行時システムのスケジューラはプロセスに割り当てられた配分時間をそのプロセスのスレッドの間で分配する。この方法により、スレッドの存在はカーネルからは全体として意識しなくなる。カーネルはただ、単一スレッドのヘビーウエイトプロセスを管理しているのみである。この方法はSunOS 4.1ライトウエイトプロセスパッケージにおいて使用されている。

他方、カーネルレベルの方法では、実行時システムはなく、スレッドはカーネルによって管理される。それゆえ、スレッドの状態情報テーブルはカーネルの中で維持される。スレッドをブロックするかもしれないすべての呼び出しは、カーネルにトラップをかけるシステム呼び出しとして実装される。スレッドがブロックすると、カーネルは別のスレッドを動作させるために選択する。選択されたスレッドは、今まで動作していたスレッドと同じプロセスに属するスレッドかもしれないし、異なるプロセスに属するかもしれない。したがって、スレッドの存在はカーネルで意識され、この方法では1レベルスケジューリン

グが使用される。

図 8.9 はスレッドパッケージの 2 つの実装方法を示す。相対的な利点と欠点を以下に述べる。

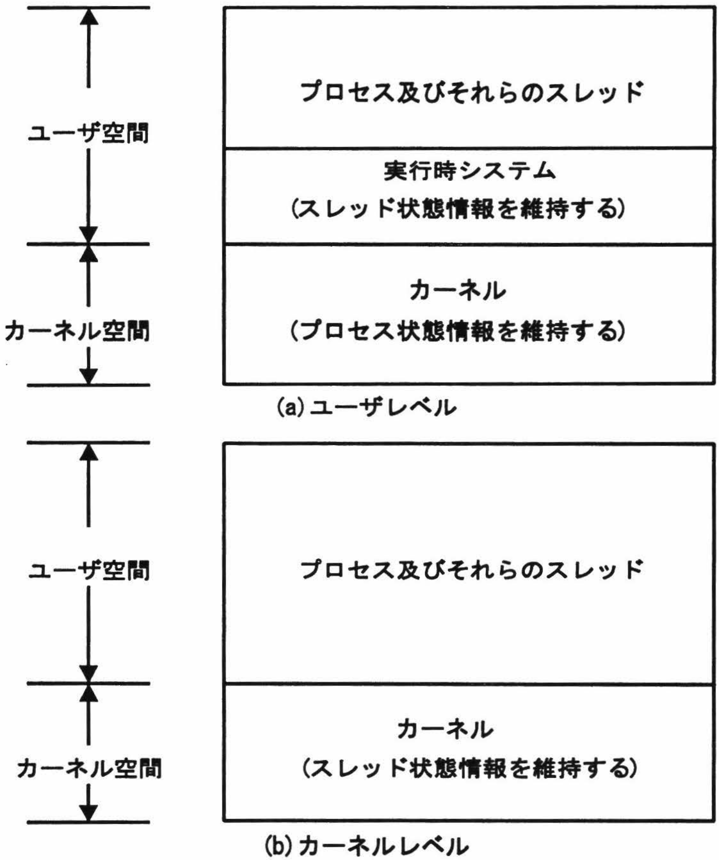


図 8.9　スレッドパッケージ実装の方法

1. ユーザレベル方法の最も重要な利点は、スレッドパッケージをスレッド機能のないオペレーティングシステムの上に実装することができることである。これはカーネルレベル方法では不可能である。カーネルレベルの方法において、スレッドの概念はオペレーティングシステムのカーネルの設計に含まれていなければならない。

2. ユーザレベル方法において、**2 レベルスケジューリング**の使用により、ユーザはプロセスのスレッドをスケジュールするために、ユーザ独自のカストマイズされたアルゴリズムを使用する柔軟性を有する。それゆえ、アプリケーションのニーズに応じて、ユーザは最も適切なスケジューリングアルゴリズムを設計し使用することができる。このようなことはカーネルレベルの方法では、カーネルの中に組み込まれた 1 レベルスケジューラを使用するので、不可能である。それゆえ、ユーザは、システム呼び出しのパラメータを通して、プロセスのそれぞれのスレッドに優先度を指定すること、及び既に実装されたスケジューリングアルゴリズムの集合から選択するという自由度をもつのみである。

3. 1 つのスレッドから別のスレッドへのコンテキストの切り替えはユーザレベル方法の方がカーネルレベル方法よりも高速である。これは、前者の方法での切り替えは実行時システムで行われ、後者の方法はカーネルへのトラップを必要とするからである。

4. カーネルレベル方法では、スレッドの状態情報テーブルはカーネルの中に維持される。この理由で、カーネルレベル方法の規模拡張性はユーザレベル方法に比べて乏しい。

5. ユーザレベル方法の重大な欠点は、配分時間単位ベースでスレッド間をCPU サイクルをタイムシェアするラウンドロビンスケジューリング方策が使用できないという点である [Tanenbaum 1995]。これは 1 つのプロセス内でのクロック割り込みを行えないことによる。それゆえ、一度スレッドが動作するために CPU が与えられると、それに割り込みをかける方法がなく、スレッドが自発的に CPU を放棄するまで動作を続ける。これはカーネルレベル方法では異なる。この方法では、クロック割り込みは周期的に発生し、カーネルはあるスレッドによって使用された CPU 時間の量を把握することができる。スレッドが割り当てられた配分時間を使用し終わると、カーネルが割り込みを行い、別のスレッドに CPU を与えることができる。

この問題を解決する単純な方法は、実行時システムを固定の時間 (例えば 0.5 秒) 毎にクロック割り込みを要求させ、制御をさせるようにすることである [Tanenbaum 1995]。実行時システムが制御権を得ると、スケジューラは、スレッドが動作を継続するべきか、他のスレッドに CPU を移すべきかを決定す

る。

6. ユーザレベル方法の別の欠点はシブロッキングステム呼び出し (ブロックするためのシステム呼び出しし) の実装に関わる。カーネルレベル方法では、ブロッキングシステム呼び出しの実装は直接的である。スレッドがそのような呼び出しを行うと、カーネルにトラップを行い、そこでスレッドは保留となりカーネルは新しいスレッドを開始させる。しかし、ユーザレベル方法では、スレッドはブロッキングシステム呼び出しを直接行うことは許されていない。これは、もしスレッドが直接的にブロッキングシステム呼び出しを行うならば、そのプロセスのすべてのスレッドが停止し、カーネルは動作させる別のプロセスをスケジュールしなければならないという理由による。そしてその場合には、スレッドを用いる基本的な目的が失われる。

この問題の対策として通常使用される方法は**ジャケット (上着) ルーチン**を使用することである。ジャケットルーチン (jacket routine) は、各ブロッキングシステム呼び出しの前に、その呼び出しがカーネルにトラップを生じるか検査する、追加のコード (プログラム) を含んでいる。呼び出しは、安全である (カーネルにトラップを発生しない) ならば、なされる。そうでなければ、スレッドは保留となり、他のスレッドが動作するためにスケジュールされる。安全の条件を検査すること及び実際の呼び出しは原子的に行われなければならない。

少しは成功例があるが、スレッドパッケージの実装において、ユーザレベルとカーネルレベルの方法の利点を組み合わせる試みは余りなされていない。例えば、FastThread パッケージ [Anderson 他 1991] 及び Psyche マルチプロセッサオペレーティングシステム [Marsh 他 1991] のスレッドパッケージはユーザレベルスレッドスケジューリングのカーネルサポートを提供する。他方、 Mach [Black 1990] はユーザレベルのコード (プログラム) がカーネルのスレッドスケジューラにスケジューリングのヒントを与える。Mach におけるスレッド機能の実装の詳細は第12章で述べる。

8.3.5 事例研究：DCE スレッド

Mach オペレーティングシステムのために開発された C Threads(C スレッド) パッケージ及び SunOS のために開発された Lightweight Processes(ライトウエイトプロセス) パッケージは商用のスレッドパッケージの 2 つの例である。加えて、スレッド設計間の互換性のために、IEEE は **P-Thread(P ス**

レッド)として知られているPOSIX (Portable Operating System Interface for Computer Environments) スレッド標準の案を作成した。この標準の2つの実装がOSF(Open Software Foundation)によってOSF/1オペレーティングシステム用に開発されたDCE Threads及びSunOS用にFree Software Foundationによって開発されたGNU Threadsである。P1003.4a POSIX標準に基づくDCE Threadパッケージを事例研究として以下に述べる。別のスレッドパッケージであるC Threadについて、第12章でMachオペレーティングシステムの一部として紹介する。

ユーザレベルの方法がDCE Threadパッケージの実装に使用される。すなわち、DCEはスレッドの生成、終了、同期、その他のためにユーザレベルライブラリ手続きを提供する。C言語で書かれたアプリケーションからスレッドサービスをアクセスするために、DCEはPOSIX標準に準拠するアプリケーションプログラムインタフェース(API)を規定する。もしDCEをサポートするシステムが本来のスレッドをサポートしていないならば、APIはアプリケーション手続きにリンクされるDCEスレッドライブラリへのインタフェースを提供する。他方、もしDCEをサポートするシステムはスレッドをサポートするオペレーティングシステムカーネルを持っているならば、DCEはこの機能を使用するために設定される。この場合、APIは、カーネルによってサポートされたスレッド機能へのインタフェースを提供する。

スレッド管理

DCE Threadパッケージはスレッドを管理し扱うための多数のライブラリ手続きを持っている。重要なものの幾つかを以下に示す。

■ **pthread_create**は呼び出しを行ったスレッドと同じアドレス空間に新しいスレッドを生成するために使用される。スレッドは親スレッドと並行で動作する。しかし、親スレッドのコードを実行する代わりに、**pthread_create**ルーチンの入力パラメータとして指定される名前の手続きを実行する。

■ **pthread_exit**は呼び出しを行ったスレッドを終了させる。このルーチンはその仕事を終了したスレッドによって呼び出される。

■ **pthread_join**はこのルーチンの引き数によって指定されたスレッドが終了するまで、呼び出しを行ったスレッドが自身をブロックするために使用される。このルーチンはUNIXの**wait**システム呼び出しに類似し、親スレッドが子スレッドの実行完了を待つために使用される。

■ **pthread_detach** は子スレッドの所有を放棄するために親スレッドによって使用される。**pthread_detach** を呼び出すことにより、親スレッドは、指定した子スレッドを **pthread_join** しない (待たない) ことを通知する。子スレッドが **pthread_exit** を呼ぶと、そのスタックとその他の情報は直ちに再生される。通常の場合、この整理動作は親スレッドが **pthread_join** を正常に終わった後に行われる。

■ **pthread_cancel** は別のスレッドを強制終了させるために使用される。

■ **pthread_setcancel** は他のスレッドを強制終了させる能力を与えたり、取り去ったりするために使用される。例えばスレッドが mutex 変数を持っていて、スレッドの強制終了が致命的結果となるような時に、pthread_setcance 機能により、スレッドが別のスレッドから強制終了させられることを防止できる。

スレッド同期

DCE Threads パッケージは**スレッド同期**のために、mutex 変数及び条件変数をサポートする。mutex 変数は、複数スレッドによる共有資源へのアクセスが同時には相互に排他的でなければならない時に使用される。他方、条件変数は、既に別のスレッドによってロックされた共有資源を、使用しているスレッドがアンロックし、待っているスレッドにシグナルがなされるまで、スレッドがブロックし待つために mutex 変数とともに使用される。

DCE Threads パッケージは次のタイプの mutex 変数をサポートする。入れ子のロックをどのように処理するかによって異なる。

1. **高速 (fast)**。高速 mutex 変数は、スレッドが既にロックされている mutex 変数をロックしようとすると、スレッドをブロックするタイプである。すなわち、高速 mutex 変数の入れ子のロック操作は許されていない。高速 mutex 変数はデッドロックを招くかもしれないことに注意されたい。例えば、もしスレッドが同じ mutex 変数を 2 回目としてロックしようとすると、デッドロックが起きる。

2. **再帰 (recursive)**。再帰 mutex 変数は既にロックされた mutex 変数をスレッドがロックすることを許容する。すなわち、再帰 mutex 変数の入れ子のロック操作は、任意の深さで許される。再帰 mutex 変数はデッドロックにならないことに注意されたい。再帰 mutex 変数をロックされた回数だけアンロックすることは、アプリケーションプログラマの責任である。

3. 非再帰 (**nonrecursive**)。非再帰 mutex 変数は、既にロックされている mutex 変数をロックすることも、スレッドがブロックすることも許容しない mutex 変数である。その代わり、既にロックされている非再帰 mutex 変数をロックしようとすると、誤りコードが戻される。非再帰 mutex 変数は高速 mutex 変数に伴うデッドロック問題を回避することができることに注意されたい。

スレッド同期のための DCE スレッド呼び出しの主な幾つかを以下に示す。

■ **pthread_mutex_init** は mutex 変数を動的に生成するために使用される。

■ **pthread_mutex_destroy** は mutex 変数を動的に削除するために使用される。

■ **pthread_mutex_lock** は mutex 変数をロックするために使用される。もし指定された mutex 変数が既にロックされているならば、この呼び出しを行ったスレッドは mutex 変数がアンロックされるまでブロックされる。

■ **pthread_mutex_trylock** は mutex 変数をロック要求するために使用される。もし mutex 変数が既にロックされているならば、その呼び出しはスレッドをロックする代わりに結果が不成功であったことを戻す。

■ **pthread_mutex_unlock** は mutex 変数をアンロックするために使用される。

■ **pthread_cond_init** は条件変数を動的に生成するために使用される。

■ **pthread_cond_destroy** は条件変数を動的に削除するために使用される。

■ **pthread_cond_wait** は条件変数を待つために使用される。呼び出しを行ったスレッドは、pthread_cond_signal 又は pthread_cond_broadcast が条件変数のために実行されるまで、ブロックされる。

■ **pthread_cond_signal** は条件変数を待っているスレッドを再開させる。もし複数のスレッドが条件変数を待っているならば、ただ 1 つのスレッドが再開され、他のスレッドは待ち続ける。

■ **pthread_cond_broadcast** は条件変数を待っているすべてのスレッドを再開させるために使用される。

相互排他が必要な別の分野は UNIX ライブラリ手続きの使用の分野である。UNIX の標準ライブラリ手続きは再帰的ではない。それゆえ、任意の時間でのスレッド切り替え時における一貫性を保つために、個別の呼び出しごとに相互

排他を提供する必要がある。

DCE はこの問題を再帰的でない UNIX システム呼び出し (ほとんどは I/O 手続き) の数だけジャケットルーチンを提供することにより解決している。ジャケットルーチンは、起こりうる問題を避けるために、システム呼び出しの前又は後にスレッドを代行して必要な動作を行う。例えば、ジャケットルーチンは、特定のサービスの呼び出しは同時には 1 つのみであることを保証する。

他の幾つかの UNIX 手続きに対して、DCE は、ライブラリに同時にはただ 1 つのスレッドのみが動作することを保証するために、1 つの全体 mutex 変数を提供している。

スレッドスケジューリング

DCE Threads パッケージは優先度ベースのスレッドスケジューリングをサポートする。これは、ユーザが個別のスケジューリングに優先度を指定でき、重要なスレッドが他のスレッドよりも優先して必要時に必要な CPU 時間を取れるようなスケジューリングアルゴリズムを使用できる。ユーザは次のスレッドスケジューリングアルゴリズムから 1 つを選ぶことができる。

1. **先入れ先出し (First In, First Out: FIFO)**: この方法では、最高優先度の待ち行列から最初の待ちのスレッドが動作するために選ばれる。選ばれたスレッドはブロック又は終了するまで実行を続ける。スレッドが実行を終了した後に、同じ方法が新しいスレッドに CPU を割り当てるために選ばれる。このアルゴリズムは低優先度スレッドの飢餓が発生することがありうる。

2. **ラウンドロビン (Round Robin (RR)**: この方法では、最高待ち行列から最初の待ちが選ばれる。しかし、この最初のスレッドが完了まで動作する代わりに、この待ち行列のすべてのスレッドは、ラウンドロビン方式で固定の配分時間の間各スレッドを動作させることによって平等の重要性を与えられる。このアルゴリズムはやはり低優先度スレッドの飢餓を起こすことがありうる。

3. **デフォルト (default)**: この方法では、すべての優先度の待ち行列上のスレッドが 1 つずつタイムスライスされ、ラウンドロビンアルゴリズムで実行される。スレッドに割り当てられた持ち時間はその優先度に応じて変化する。すなわち、優先度が高いほど、長い持ち時間が与えられる。このアルゴリズムにおいて、すべてのスレッドが動作する機会が与えられるので、飢餓は生じない。

次のシステム呼び出しにより、ユーザはスケジューリングアルゴリズムを選択できる。

■ **pthread_setscheduler** はスケジューリングアルゴリズムを選択するために使用される。

■ **pthread_getscheduler** は現在どのスケジューリングアルゴリズムが働いているかを知るために使用される。

■ **pthrad_setprio** はスレッドのスケジューリング優先度を設定するために使用される。

■ **pthread_getprio** はスレッドのスケジューリング優先度を知るために使用される。

シグナル処理

シグナルは、スレッドの実行中にセグメンテーション違反や浮動小数点例外の例外条件の発生が理由で、又はキーボードの該当するキーを打つことにより動作中のプロセスにユーザが意図的に割り込みを行うなどの外部割り込みが理由で生成される。DCE において、例外条件は、それが発生したスレッドによって処理される。しかし、外部割り込みは、関連するすべてのスレッドによって処理される。すなわち、外部割り込みが発生するとき、スレッドパッケージは、その割り込みを待っているすべてのスレッドにそれを渡す。

DCE はまた、伝統的な方法であるシグナルハンドラを捕捉するという代わりにシグナル処理に使用される POSIX の **sigwait** 及び **sigaction** サービスを含んでいる。これらのサービスはシグナルハンドラとは異なるレベルで動作するが同じ結果を達成することができる。**Sigwait** サービスによって、特定の割り込みベースのシグナルの集合 (非同期シグナルとしても知られている) の 1 つが渡されるまで、スレッドはブロックできる。他方、**sigaction** サービスによって、例外ベースのシグナル (同期シグナルとしても知られている) を捕捉するために、スレッドごとのハンドラを実装することができる。

誤り処理

標準の P1003.4a P-Threads 呼び出しと同様に UNIX のシステム呼び出しはグローバル変数 **errno** をセットし、−1 を戻すことにより、誤りを報告する。この誤り処理の方法において、先の **errno** 値が発生したスレッドによって取り込まれ処理される前に、他のスレッドで誤りが起きてグローバル変数 **errno** が

上書きされると、ある誤りは紛失するかもしれない。この問題の対策として、DCEにおける各スレッドは、スレッドごとの誤り状態を格納するための個別の**errno**変数を持っている。この変数はスレッド切り替えの時、他のスレッドに関する項目とともに、保存及び復元される。DCEの誤り処理インタフェースによって、プログラマはこの変数を調べることができる。誤り処理のためにDCEにおいて使われるかもしれない別の方法は、誤りが発生した時にシステム呼び出しに例外を起こさせることである。

8.4 まとめ

本章は分散オペレーティングシステムにおける2つの重要なプロセス管理の概念であるプロセス移動とスレッドについて述べた。

プロセス移動は、分散システムにおいて1つのノードから別のノードへの透過的な再配置について取り扱う。プロセスは実行開始前又は実行途中で再配置されることがある。前者は先取り権なしのプロセス移動と呼ばれ、後者は先取り権付きのプロセス移動として知られている。先取り権付きプロセス移動は、先取り権なしプロセス移動よりも高コストである。新しいノードへのプロセス移動に伴うプロセスの状態の扱いが実行開始後では、複雑になるからである。

プロセス移動方策は、プロセス移動元である送信元ノードの選択、プロセス移動先である宛先ノードの選択、及び移動対象プロセスの選択を扱う。これらの選択決定は、プロセス移動方策に使用される適切な全体スケジューリングアルゴリズムによって行われる。他方、プロセス移動機構は、送信元ノードから宛先ノードへのプロセスの実際の転送並びに、移動の間、及び後の関連するメッセージの転送及び処理を扱う。先取り権付きプロセス移動のために、通常使用される方法は、送信元ノードでプロセスを凍結させ、宛先ノードにプロセスの状態情報を転送し、それからこの状態情報を用いて宛先ノード上でプロセスを再開始するという方法である。

プロセスを移動させるコストは、そのアドレス空間を移動させるための時間によってほぼ定まる。全体凍結、事前転送、及び参照時転送はアドレス空間転送のために、現存するシステムで使用されている機構である。現存するシステムのメッセージ転送に使用されている、別の機構は、メッセージ再送機構、発生元サイト機構、リンク横断機構、及びリンク更新機構である。異なるノードに移動され、配置されるかもしれないプロセスとサブプロセスとの間の通信のための2つの異なる機構は、論理ホストの概念及びホームノード又は発生元サ

イトの概念である。

異種システムにおけるプロセス移動は同種システムにおけるよりも、送信元ノードのデータフォーマットから宛先ノードのデータフォーマットにデータ変換する必要があるので、複雑となる。外部データ表現機構は、この変換過程のソフトウェアの複雑性を減少する手段となりうる。文字データ、整数、浮動小数点数のような各種のデータタイプのうちで、浮動小数点数の扱いは特別の注意が必要である。

プロセス移動機能の現存する実装は、先取り権付きプロセス移動は、当初の予想以上に大きいオーバヘッドと複雑さはあるが、実現可能であることを示している。移動のコストゆえに、何人かのシステム設計者は、反対者もいるが、これは実現可能な代替案ではないと結論している。しかしこの話題は、関連する研究を伴い、今でも活発な研究領域である。

スレッドは並列処理によりアプリケーション性能を改善するために、ますます一般的になっている方法である。スレッド機能をもつオペレーティングシステムにおいて、プロセスは1つのアドレス空間と1つ又はそれ以上の制御のスレッドからなっている。プロセスの各スレッドは各自のプログラムカウンタ、各自のレジスタ状態、及び各自のスタックをもっている。しかし、プロセスのすべてのスレッドは同じアドレス空間を共有する。スレッドはしばしばライトウエイトプロセスと言われ、伝統的なプロセスはヘビーウエイトプロセスと言われる。

スレッドを使用する主な動機は、コンテキスト切り替え時間を最小にすることであり、最小のオーバヘッドで、CPUが1つの計算のユニットから別の計算のユニットへ切り替えることができる。別の重要なスレッドの動機は、逐次実行とブロッキングシステム呼び出しを組み合わせることにより並列処理ができることである。スレッドの使用は、共有のアドレス空間を使用する一式のスレッドは、多くのアプリケーションをプログラムするための最も自然な方法であるという事実によっても起因している。

プロセスのスレッドを組織化するための3つのよく使用される方法は、ディスパッチャワーカモデル、チームモデル、及びパイプラインモデルである。

スレッド関連操作のためにユーザに提供されているプリミティブの集合はスレッドパッケージを形成すると言われる。スレッドパッケージを設計する際の重要事項の幾つかは、スレッドの生成、終了、同期、及びスレッドのスケジューリングであり、並びにシグナル及び誤り処理である。

スレッドパッケージはユーザ空間又はカーネル空間のどちらかに実装される。

両者ともに、それぞれ利点もあれば県展欠点もある。

8.5　演習問題

8.1 先取り権付き、及び先取り権なしのプロセス移動の区別を述べよ。それらの相対的な利点及び欠点は何か。読者が分散システムのためのプロセス移動機能を設計する場合を想定する。先取り権付き、又は先取り権なしのプロセス移動機能を設計する際に、どのような要素を考慮すべきであるか。

8.2 送信元ノード上で移動プロセスを凍結する際、及び宛先ノード上で再開する際の主な問題点は何か。これらの問題を取り扱うための方法を提案せよ。

8.3 次の 2 つの処理の実装の主な類似点と相違点は何か。

(a) 高い優先度のプロセスを実行するためにプロセスに割り込みを行うこと、及び同じノード上で、ある時間の後に、割り込まれたプロセスを再開させること

(b) 移動プロセスを凍結すること、及び異なるノード上で、それを再開すること。

8.4 先取り権付きプロセス移動機能をサポートするという観点から、状態無しのファイルサーバと状態有りのファイルサーバのどちらが好ましいか。また、その理由を述べよ。

8.5 移動プロセスは、移動の後に宛先ノード上で、再開するとき、送信元上でもっていたのと同じプロセス識別子が与えられる。これは必要であるか。その答えの理由を述べよ。

8.6 プロセス移動の主なコストは、そのアドレス空間の転送に要する時間である。このコストを最小にするために使用できる方法を提案せよ。

8.7 ある分散システムは DSM(Distributed Shared Memory) をサポートしている。このシステムのためにプロセス移動機能を設計するために使用する適切なアドレス空間転送機構を提案せよ。

8.8 次の目標をもつプロセス移動機能のためには、本章で述べたアドレス空間転送機構のどの 1 つ (以上) が適切であるか。

(a) 高性能が主な目標

(b) 高信頼性が主な目標

(c) プロセス移動方策の効率が主な目標

(d) 単純な実装が主な目標

(e) プロセス移動方策の信頼性及び効率の両者が重要な目標

もし特定の場合に、1 つより多くの機構が適切であるならば、読者はどちら

を好むか、またその理由は何か。

8.9 次の目標をもつプロセス移動機能のためには、本章で述べたメッセージ転送機構のどの1つ(以上)が適切であるか。

(a) 透過性が主な目標

(b) 信頼性が主な目標

(c) 性能が主な目標

(d) 単純な実装が主な目標

もし特定の場合に、1つより多くの機構が適切であるならば、読者はどちらを好むか、またその理由は何か。

8.10 次の目標をもつプロセス移動機能のためには、本章で述べたコプロセス間の通信を扱うための機構のうちのどれが適切であるか。

(a) 性能が主な目標

(b) 信頼性が主な目標

(c) 単純な実装が主な目標

もし特定の場合に、1つより多くの機構が適切であるならば、読者はどちらを選択するか、またその理由は何か。

8.11 異分散システムのためのプロセス移動機能の設計に含まれる主な問題点は何か。

8.12 分散システムのプロセス移動機能は、1つのノードから他のノードへの自由なプロセスの移動を許容しない。どのプロセスがシステムの他のノードへ移動させられるかに関してある種の制約をもっている。このような制約を課している理由は何か。

8.13 プロセス移動機能の設計において使用される外部データ表現機構はいつ使用されるべきであるか。読者がプロセス移動機能のための外部データ表現フォーマットを設計する場合を想定する。読者が設計決定を行う際に重要な要素は何か。

8.14 ある分散システムは3つのタイプのプロセッサA、B、及びCをもっている。プロセッサタイプA、B、及びCにおける浮動小数点数の指数に使用されるビット数が、それぞれ8、12、及び16であり、仮数に使用されるビット数が、それぞれ16、32、及び64である。このシステムにおいて、どのタイプからどのタイプへの移動が許容され、どのタイプからどのタイプへの移動が許されないか。その答えと理由を述べよ。

8.15 プロセス移動の利点と欠点の幾つかを挙げよ。

8.16 プロセスがCPU使用の基本的単位であるオペレーティングシステムに

おいて、あるプロセスを他のプロセスから保護する機構が提供される。スレッドがCPU使用の基本的単位であるオペレーティングシステムにおいて、あるスレッドを他のスレッドから保護する類似の機構を提供する必要があるか。その答えと理由を述べよ。

8.17 スレッドの概念が、よりよい性能のために分散オペレーティングシステムにおいて、しばしば使用される。この概念が、他のマルチプロセッサオペレーティングシステムや従来の集中型のタイムシェアリングシステムに対するオペレーティングシステムにおいて、性能を改善するために有用であるか。その答えと理由を述べよ。

8.18 スレッドとプロセスの主な相違点と類似点を挙げよ。

8.19 複数プロセスの代わりにスケジューリングを使用することの主な利点と欠点は何か。スレッドの使用により利益を受けるアプリケーションの例及び利益を受けないアプリケーションの例を述べよ。

8.20 分散システムにおいて、並列処理は性能を改善し、ブロッキングシステム呼び出しのはプログラミングを容易にする。スレッドの概念がどのように、両者の利点を組み合わせるかについて説明せよ。

8.21 スレッドによって実現される並行性から利益を受けるようにサーバプロセスをどのように設計するのかを示す例を挙げよ。次に同じく利益を受けるようにクライアントプロセスをどのように設計するのかを示す例を挙げよ。

8.22 次のそれぞれに対する適切な例を挙げよ。

(a) プロセスがディスパッチャワーカモデルで組織化された複数スレッドを使用するアプリケーション

(b) プロセスがチームモデルで組織化された複数スレッドを使用するアプリケーション

(c) プロセスがパイプラインモデルで組織化された複数スレッドを使用するアプリケーション

8.23 あるファイルサーバは次のように動作する；

(a) ファイルアクセスのためのクライアントからの要求を受け付ける。

(b) 次に、サーバは自己が保持しているキャッシュの中のデータを用いて要求に応えようとする。

(c) もし要求がキャッシュされたデータを用いてサービスできないならば、データを求めてディスクサーバに要求を行い、応答が受信されるまで休眠する。応答を受信すると、受信したデータをキャッシュし、クライアントの要求をサービスする。

キャッシュのヒット率を0.7とする。すなわち、すべての要求の70 %がキャッシュされたデータを用いてサービスされ、ディスクサーバへのアクセスはすべての要求の30 %のサービスのために必要である。また、キャッシュヒットの場合、要求サービス時間は、20ミリ秒であり、キャッシュミスの場合、要求サービス時間は100ミリ秒であるとする。ファイルサーバが次のように実装されている場合、毎秒あたりどれだけの要求がサービスされ得るか。

(a) 単一スレッドのプロセス

(b) 複数スレッドのプロセス

スレッド切り替え時間は無視できるものとする。

8.24 ハンドオフスケジューリングと関連スケジューリングの区別を述べよ。読者の意見では、そちらがスレッドパッケージにより望ましいか。またその理由を述べよ。

8.25 動的に可変の配分時間の柔軟性及びハンドオフスケジューリングを提供するスレッドスケジューリングアルゴリズムのための擬似コードを作成せよ。

8.26 マルチスレッド環境におけるシグナルを扱う際の主な問題点は何か。これらの問題のそれぞれを扱うための方法を述べよ。

8.27 ユーザ空間及びカーネル空間にスレッドパッケージを実装する場合の相対的な利点及び欠点について述べよ。

8.28 あるコンピュータのオペレーティングシステムは、CPU使用の基本的単位として、プロセスを使用している。すなわち、スレッドをサポートしていない。このコンピュータシステムにおいて、オペレーティングシステムのカーネルを修正することなく、スレッド機能は提供されることができるであろうか。できないならその理由を、できるならその理由を述べよ。

参考文献

[Agrawal and Ezzat 1987] Agrawal, R., and Ezzat, A. K., "Locatoin Independent Remote Execution in NEST," IEEE Transaction on Software Engineering, Vol. SE-13, No. 8 (1987).

[Alonso and Kyrimis 1988] Alonso, R., and Kyrimis, K., "A Process Migration Implementaion for a UNIX System," In: Proceedings of the Winter 1988 Usenix Conference, Usenix Association, Berkeley, CA (February 1988).

[Anderson et al. 1991] Anderson, T. E., Bershad, B. N., Lazowska, E. D., and Levy, H. M., "Scheduler Activations: Effective Kernel Support

for the Use-Level Management of Parallelism," In: proceedings of the 13th ACM Symposium on Operating System Principles, Association for Computing Machinery, New York, NY, pp. 95-109 (1991).

[Artsy and Finkel 1989] Artsy, Y., and Finkel, R., "Designing a Process Migration Facility," IEEE Computer, Vol. 22, pp. 47-56 (1989).

[Black 1990] Black, D., "Scheduling Support for Concurrency and Parallelism in the Mach Operationg System," IEEE Computer, Vol. 23, pp.35-43 (1990).

[Butterfield and Popek 1984] Butterfield, D. A., and Popek, G. J., "Network Tasking in the LOCUS Distributed UNIX System," In: Proceedings of the Summer 1984 Usenix Conference, Usenix Association, Berkeley CA, pp. 62-71 (June 1984).

[Cheriton 1988] Cheriton, D. R., "The V Distributed System," Communications of the ACM, Vol. 31, No. 3, pp. 314-333 (1988).

[Coulouris et al. 1994] Coulouris, G. F., Dollimore, J., and Kindberg, T., Distributed Systems Concepts and Design, 2nd ed., Addison-Wesley, Reading, MA (1994).

[Douglis and Ousterhout 1987] Dougis, F., and Ousterhout, J., "Process Migration in the Sprite Operating System," In: Proceedings of the 7th International Conferene on Distributed Computing Systems, IEEE, New York, NY, pp. 18-25 (September 1987).

[Douglis and Ousterhout 1991] Douglis, F., and Outerhout, J., "Transparent Process Migration: Design Alternatives and the Sprite Implementatin," Software—Practice and Experience, Vol. 21, pp. 757-785 (1991).

[Draves et al. 1991] Draves, R. P., Bershad, B. N., Rashhid, R. F., and Dean, R. W., "Using Continuations to Implement Thread Management and Communication in Operating Systems," In: Proceedings of the 13th ACM Symposium on Operating System Principles, Association for Computing Machinery, New York, NY, pp. 122-136 (1991).

[Ferrari and Sunderam 1995] Ferrari, A., and Sunderam, V. S., "TPVM: Distributed Concurrent Computing with Lightweight Processes," In: Proceedings of the 4th International Symposium on High Performance Distributed Computing (August 1995).

[Goscinski 1991] Goscinski, A., "Distributed Operating Systems, The Logical Design," Addison-Wesley, Reading, MA (1991).

[Huang et al. 1995] Huang, C., Huang, Y., and McKinley, P. K., "A Thread-Based Interface for Collective Communication on ATM Networks," In: Proceedings of the 15th International Conference on Distributed Computing Systems, IEEE New York, NY (May-June 1995).

[Hunter 1988] hunter, C., "Process Cloning: A System for Duplicating UNIX Processes," In: Proceedings of the Winter 1988 Usenix Conference, Usenix Association, Berkeley, CA (February 1988).

[Jul 1989] Jul, E., "Migration of Light-Weight Processes in Emerald," TCOS Newsletter, Vol. 3, No. 1, pp. 20-23 (1989).

[Jul et al. 1988] Jul. E., Levy, H., Norman, H., and Andrew, B., "Fine-Grained Mobility in the Emerald System," ACM Transactions on Computer Systems, Vol. 6, No. 1, pp. 109-133 (1988).

[Kingsbury and Kline 1989] Kingsbury, B. A., and Kline, J. T., "Job and Process Recovery in a UNIX-Based Operating System," In: Proceedings of the Winter 1989 Usenixs Conference, Usenix Associaton, Berkeley, CA, pp. 355-364 (1989).

[Litzkow 1987] Litzkow, M. J., "Remote UNIX—Turning Idle Workstations into Cycle Servers," In: Proceedings of the Summer 1987 Usenix Conference, Usenix Association, Berkeley, CA (June 1987).

[Lo1989] Lo. V. M., "process Migration for Communication Performance," TCOS Newsletter, Vol. 3, No. 1, pp. 28-30 (1989). **[Lockhart 1994]** Lockhart, Fr., H. W¿, OSF DCE: Guide to Developing Distributed Applications, IEEE Computer Society Press, Los Alamitos, CA (1994).

[Maguire and Smith 1988] Maguire, Jr., G. Q. and Smith, J. M., "Process Migration: Effects on Scientific Computation," ACM-SIGPLAN Notices, Vol. 23, No. 3, pp. 102-106 (1988).

[Mandelberg and Sunderam 1988] Mandelberg, K.I., and Sunderam, V. S., "Process Migration in UNIX Networks," In: Proceedings of the Winter 1988 Usenix Conference, Usenix Association, Berkeley, CA (February 1988).

[Marsh et al. 1991] Marsh, B. D., Scott, M. L., LeBlanc, T. J., and Markatos, E. P., "First-Class Use-Level Threads," In: Proceedings of the 13th ACM Symposium on Operating System Principles, Association for Computing Machinery, New York, NY, pp. 110-121 (1991).

[Mullender et al. 1990] Mullendr. S. J., Van Rossum, G., Tanenbaum, A. S., Van Renesse, R., and Van Staverene, H., "Amoeba: A Distriubted Operating System for the 1990s" IEEE Computer Vol.23, No.5, pp. 44-53 (1990).

[Nutt 1991] Nutt, G. J., Centralized and Distributed Operagting Systems, Prentice-Hall, Englewood Cliffs, NJ (1991).

[Popek and Walker 1985] Popek, G. J., and Walker, B. J., The Locus Distirbuted System Architecture, MIT Press, Cambridge, MA (1985).

[Powell and Miller 1983] Powell, M. L., and Mller, B. P., "Process Migration in DEMOS/MP," In: Proceedings of the 9th ACM Symposium on Operating System Principles, Association for Computing Machinery, New York, NY, pp. 110-119 (November 1983).

[Rosenberry et al. 1992] Rosenberry, W., Kenney, D., and Fisher, G., OSF DISTRIBUTD COMPUTING ENVIRONMENT¡ Understanding DCE, O'Reilly & Associates, Sebastopol, CA (1992).

[Schwan et al. 1991] Schwan, K., Zhou, H., and Gheith, A., "Real-Time Threads," ACM-SIGOPS Operating Systems Review, Vol. 25, No. 4, pp.35-46 (1991).

[Sinha et al. 1991] Sinha, P. K., Park, K., Jia, X., Shimizu, K., and Maekawa, M., "Process Migratino Mechanism in the Galaxy Distributed Operating System," In: Proceedings of the 5th Internatinoal Parallel Processing Symposium, IEEE, New York, NY, pp. 611-618 (April 1991).

[Smith 1988] Smith, J. M., "A Survey of Process Migration Mechanisms," ACM-SIGOPS Operating Systems Review, Vol. 22, pp. 28-40 (July 1988).

[Stalling 1995] Stalling, W., Operating Systems, 2nd ed., Prentice-Hall, Englewood Cliffs, NJ (1995).

[Tanenbaum 1995] Tanenbaum, A. S., Distributed Operating Systems, Prentice-Hall, Englewood Cliffs, NJ (1995).

[Theimer et al. 1985] Theimer, M. M., Lantz K. A., and Cheriton, D. R., "Preemptable Remote Execution Facilities for the V System," In: Proceedings of the 10th ACM Symposium on Operating System Principles, Association for Computing Machinery, New York, NY, pp. 2-12 (December 1985).

[Thekkath and Eggers 1994] Thekkath. R., and Eggers. S. J., "Impact of Sharing-Based Thread Placement on Multithreaded Architectures," In: Proceedings of the 21st International Symposium on Computer Architecture, Association for Computing Machinery, New York, NY, pp. 176-186 (1994).

[Walker and Mathews 1989] Walker, B. F., and Mathews, R. M., "Process Migration in AIX's Transparent Computing Facility (TCF)." TCOS Newsletter, Vol. 3, No. 1, pp.5-7 (1989).

[Zayas 1987] Zayas, E. R., "Attacking the Processs Migratin Bottlenck," In: Proceedings of the 11th ACM Symposium on Operating Systems Principles, Association for Computing Machinery, New York, NY, pp.13-22 (November 1987).

インターネット参考文献

プロセス移動に関する参考文献は、以下のところで見つけることができる。
ftp:ftp.cs.umanitoba.ca/pub/bibliographies/Distributed/migrate.html
ftp:ftp.cs.umanitoba.ca/pub/bibliographies/Distributed/dshell.html

第9章

分散ファイルシステム

9.1 はじめに

コンピュータシステムでは、ファイル (**file**) とは明示的に作成されたオブジェクトであり、システムの一時的な障害を免れることができ、明示的に消去されるまで永続的である。ファイルを使う2つの主な目的を以下に示す。

1. **情報の永続記憶 (permanent storage of information)**。これは、磁気ディスクのような2次記憶メディアへのファイルの格納によって達成される。

2. **情報の共有 (sharing of information)**。ファイルは、情報共有の自然で容易な方法を提供する。つまり、ファイルはあるアプリケーションによって作られ、別のアプリケーションによって後で共有することができる。

ファイルシステム (file system) はオペレーティングシステムの1つのサブシステムであり、ファイルの組織化、格納、取り出し、名前付け、共有、そして保護などのファイル管理機能を実行する。ファイルシステムでは、ファイル抽象化を特徴づけている、空間の確保や2次記憶装置のレイアウトなどの詳細をプログラマに隠蔽する操作群を利用できるように設計されている。したがって、ファイルシステムは、記憶装置の抽象化を行っており、記憶装置に情報を格納したり取り出したりするのに便利な機構となっている。

分散ファイルシステム (distributed file system) は、利用者に分散システムの同様な抽象化を行っており、分散環境におけるファイルの利用を便利なも

のにしている。しかしながら、分散ファイルシステムの設計と実装は、利用者と記憶装置とが物理的に散在していることから、既存のファイルシステムより複雑である。

単一プロセッサシステムでのファイルシステムによる永続的な記憶や情報の共有という利点に加えて、分散ファイルシステムは以下のような機能をサポートする。

1. **遠隔情報共有 (remote information sharing)**:分散ファイルシステムは、ファイルの位置に関わらずシステムのどのノード上のプロセスによっても透過的にアクセスされることが可能である。したがって、あるノードのプロセスは、他のノードで動作している幾つかの他のプロセスによって後にアクセスできるファイルを作ることができる。

2. **利用者移動性 (user mobility)**:分散システムでは、利用者移動性とは、利用者が特定のノードで作業することが強制されておらず、様々な時間に様々なノードで作業する自由度を持つことを意味している。この性質は、ノード障害に対処するため、様々な時間に様々なノードで作業することが必要な何人かの利用者の仕事の性質に対応するため、そしてワークステーションが共通のプールのように管理されている環境において、どの複数のノードをも利用可能とするために必要とされる。分散ファイルシステムは、2次記憶装置の物理的な再配置を行うことなく、通常異なる時間で異なるノードで作業することを可能としている。

3. **可用性 (availability)**:耐障害性の向上のために、ファイルはシステム内の1つ以上のノードに一時的な障害が発生した場合でも利用可能である必要がある。このために、分散ファイルシステムはシステムの異なるノードに1つのファイルの複数のコピーを配置する。各コピーは、ファイルの**複製 (replica)**と呼ばれる。理想的な設計では、複数のコピーの存在も、その位置についてもクライアントには隠蔽される。

4. **ディスクレスワークステーション (diskless workstation)**:ワークステーションの他の部品に比べて、ディスクドライブは相対的に高価である。更に、ワークステーションは物理的に利用者のすぐ近くに隣接しておかれることが多いので、ディスクドライブから発生する騒音や熱はワークステーションに関して敬遠される要素である。そのため、ディスクレスワークステーションはより

経済的で、静かで、そして熱の発生もない。分散ファイルシステムは、その透過的な遠隔ファイルアクセス機能を使って、システム内でディスクレスワークステーションの利用を可能としている。

分散ファイルシステムでは、典型的に以下の 3 つのタイプのサービスを提供している。それぞれは、分散ファイルシステムの部品と考えられる。

1. **記憶装置サービス (storage service)**: これは、ファイルシステムのファイルを格納するのに使われている 2 次記憶装置の領域の確保や管理を扱う。データを格納したり取り出したりする操作を提供することにより、記憶装置の論理的なビューを提供する。ほとんどのシステムは、ファイルに対する 2 次記憶装置として磁気ディスクを使っている。したがって、記憶装置サービスは、**ディスクサービス (disk service)** としても知られている。更に、幾つかのシステムではディスクの空間を固定サイズのブロック単位で確保することから、記憶装置サービスを**ブロックサービス (block service)** ということもある。

2. **純粋ファイルサービス (true file service)**: これは、ファイル内のデータをアクセスしたり変更したりする操作やファイルの作成、削除などの、個々のファイルに対する操作に関するものである。これらのファイル操作プリミティブを正しくそして効率的に実行するには、純粋ファイルサービス部品の典型的な設計項目として、ファイルアクセス機構、ファイル共有セマンティクス、ファイルキャッシュ機構、ファイル複製機構、並行制御機構、データ一貫性と複数のコピーの更新プロトコル、そしてアクセス制御機構がある。記憶装置サービスを純粋ファイルサービスから分離したことにより、異なる記憶方式や異なる記憶媒体を単一のファイルシステムに組み合わせることができる。

3. **名前サービス (name service)**:これは、テキストの名前とファイルやファイルへの参照、つまりファイル ID との対応を提供する。第 10 章で述べるように、ファイル ID は人間の利用者にとって覚えたり使ったりするのに扱いにくいので、テキストの名前が必要とされる。ほとんどのファイルシステムでは、この対応を行うのにディレクトリを使っている。したがって、名前サービスは**ディレクトリサービス (directory service)** とも呼ばれる。ディレクトリサービスは、ディレクトリの作成や削除、ディレクトリへの新しいファイルの追加、ディレクトリへの新しいファイルの追加、ディレクトリからファイルの削除、

ファイルの名前の変更、あるディレクトリから別のディレクトリへのファイルの移動など、ディレクトリに関連する操作の実行に責任を持っている。

分散ファイルシステムの記憶装置サービスの設計と実装は、集中ファイルシステムの記憶装置サービスと同じである。記憶装置サービスの詳細について興味のある読者は、オペレーティングシステムの良い著書 [Tanenbaum 1987, Silberschatz and Galvin 1994] を参照されたい。名前サービスの設計と実装の詳細は次の章に述べる。したがって、この章では分散ファイルサービスの中の純粋なファイルサービスの設計と実装について主に扱う。

9.2 良い分散ファイルシステムに必要な機能

良い分散ファイルシステムは、以下の機能を持っている必要がある。

1. **透過性 (transparency)**: 以下の4つのタイプの透過性が望まれる。

■**構造透過性 (structure transparency)**: 必須ではないが、性能、拡張性、そして信頼性の理由から、分散ファイルシステムは通常複数のファイルサーバを利用する。各ファイルサーバは、通常ユーザプロセスで時にはカーネルプロセスであり、それが動作しているノードの2次記憶装置 (ファイルの格納に使われる) の制御を受け持っている。複数のファイルサーバでは、ファイルサーバの複数化が分散システムのクライアントにとっては透過である必要がある。特に、クライアントはファイルサーバの番号や位置そして記憶装置はクライアントが知るべきではない。理想的には、分散ファイルシステムはそのクライアントに集中型の時分割型オペレーティングシステムによって提供される従来のファイルシステムのように見える必要がある。

■**アクセス透過性 (access transparency)**: ローカルと遠隔のファイルは共に同じ方法でアクセスできなければならない。つまり、ファイルシステムインタフェースは、ローカルと遠隔のファイルについて区別するべきではない。そして、ファイルシステムは自動的にアクセスされたファイルの位置を特定し、クライアントサイトにデータを転送するように手配する必要がある。

■**名前透過性 (naming transparency)**: ファイルの名前はファイルがどこに存在しているかのヒントを与えないものである必要がある。更に、ファイルはその名前の変更無しに、分散システムの1つのノードから別のノードに移動

できる必要がある。

■複製透過性 (replication transparency): もし、ファイルが複数のノードに複製されているなら、複数のコピーが存在していること、及びそれらの位置はクライアントには隠されている必要がある。

2. **利用者移動性 (user mobility)**: 分散システムにおいて、利用者は特定のノードで作業することが強制されることなく、様々な時間に様々なノードで柔軟に作業することができなければならない。更に、ファイルシステムの性能の特性においては、常に利用しているワークステーションとは異なる別のワークステーションでファイルをアクセスした場合に劣化すべきではない。利用者移動性をサポートする 1 つの方法は、その利用者がログインするときにログインする場所に利用者の環境 (つまり使用者のホームディレクトリ) を自動的に移動することである。

3. **性能 (performance)**: ファイルシステムの性能は、通常クライアントの要求を満足するのに必要な平均時間として測定される。集中型のファイルシステムでは、この時間はファイルが格納されている 2 次記憶装置のアクセス時間と CPU 処理時間を含んでいる。しかし、分散ファイルシステムでは、この時間にはファイルが遠隔にある場合ネットワーク通信のオーバヘッドも含んでいる。分散ファイルシステムの必要な性能を定量的に示すのは難しいが、集中型のファイルシステムと比較できる程度である必要がある。利用者は、性能を向上させるためにファイルの置き場所を明示する必要性を感じるようなことがあってはならない。

4. **簡潔さと使いやすさ (simplicity and ease of use)**: 分散ファイルシステムの簡潔さと使いやすさには幾つかの課題が影響している。最も重要な課題は、分散ファイルシステムのセマンティクスが理解しやすいということである。これは、ファイルシステムの利用者インタフェースは簡潔であり、そしてできるだけコマンドの数は少ないほうが良いことを意味する。理想的な設計では、分散ファイルシステムのセマンティクスは既存の集中型で時分割型のファイルシステムと同じであるべきである。もう 1 つの使いやすさについての重要な点は、ファイルシステムは様々な種類のアプリケーションをサポートできるようにしなければならないということである。

5. 拡張性 (scalability): 新しいマシンの追加や、2つのネットワークを互いに接続することによるネットワークの拡張が日常的に行われることから、時間に伴う分散システムの拡張は避けられない。したがって、良い分散システムはシステム内のノードや利用者の増加に容易に追従するように設計されるべきである。つまり、このような拡張がサービスの重大な障害やユーザに対する性能のかなりの損失を生じるべきではない。まとめると、拡張性のある設計は、高いサービス負荷に耐え、利用者の増加に適応し、追加される資源の統合を簡単にすることが必要である。

6. 高可用性 (high availability): 分散ファイルシステムは、通信リンクの障害、マシン障害、そして記憶装置のクラッシュなど、1つ以上の要素の故障によって部分的に障害が発生したときでも、継続して機能しなければならない。部分的な障害が発生した場合、ファイルシステムは性能、機能、そしてその両者の低下を招くかもしれない。しかし、その低下は、例えば故障した要素に比例した量となっているべきである。例えば、障害が利用者の小グループに対してのみ一時的なサービスの停止を生じることは、よく受け入れられることである。

高可用性と拡張性は相互に関連した特性である。両方の特性は、制御とデータが分散している設計を必要としている。なぜなら、集中型制御装置や集中型データ格納庫のような集中型のものは、厳しい一点障害と性能ボトルネックを生じるからである。したがって、高可用性で拡張性のある分散ファイルシステムは複数のそして独立なファイルサーバを持ち、複数の独立な記憶装置を制御することが必要である。複数のサーバにおけるファイルの複製は、高可用性を提供するための主な機構である。

7. 高信頼性 (high reliability): 良い分散ファイルシステムでは、格納されたデータの失われる確率は実用的なレベルに最小化されなければならない。つまり、利用者がシステムの信頼性の低さのために自分のファイルのバックアップをとる必要性を感じるべきではない。むしろ、ファイルシステムは元のファイルが失われるような事態が生じたときに利用できるように、重要なファイルのコピーを自動的に取っておく必要がある。安定記憶は、高信頼性のために幾つかのファイルシステムで一般的に使われている技術である。

8. データ一貫性 (data integrity): ファイルは複数の利用者によって共有さ

れることが多い。共有ファイルに対して、ファイルシステムは格納されたデータの一貫性を保証しなければならない。つまり、ファイルアクセスが競合している複数の利用者による並行のアクセス要求は、何かの形式による並行制御機構を利用して適切な同期が行われる必要がある。原子トランザクションは、ファイルシステムがデータ一貫性のためにしばしば提供する高レベルの並行制御機構である。

9. **安全性 (security)**: 分散ファイルシステムは、安全でなくてはならず、それによって利用者は自分たちのデータのプライバシを守ることができる。必要な安全機構は、ファイルシステムに格納されている情報を認証されていないアクセスから守るために実装されている必要がある。更に、ファイルをアクセスする権利を渡すことも、安全に行われなければならない。つまり、権利の受信者は彼又は彼女がそれを更に別の人に渡すことを認められていなければ渡せないようにするべきである。

大規模分散システムでの結論は、安全に対するいい加減な態度は認めることはできないということである。基本的な疑問は、だれが安全を強制するかである。これについては、一般的な設計原理は、可能な限り少ない部分の一貫性に依存する安全性を持つシステムは、それが拡張しても安全をより維持できるということである。

10. **異機種性 (heterogeneity)**: 大規模化の結果、分散システムにとって異機種性は避けられない。異機種分散システムは、利用者に対して異なるアプリケーションに対する異なるコンピュータプラットフォームを利用する柔軟性を提供する。例えば、利用者はシミュレーションにはスーパコンピュータを利用し、文書処理には Macintosh を利用し、そしてプログラム開発には UNIX ワークステーションを利用することもある。利用者の各種のプラットフォームを越えた共有データへの容易なアクセスは、操作性をかなり向上させる。そのため、分散ファイルシステムは各種のワークステーションが分散ファイルシステムを通してファイルの共有に参加できるように設計されなければならない。ファイルシステムにおける別の異機種性は、幾つかの異なる記憶媒体を適応させることである。したがって、分散ファイルシステムは、比較的簡単な手順で新しいタイプのワークステーションや記憶媒体を統合できるように設計されなければならない。

9.3 ファイルモデル

様々なファイルシステムは様々なファイルの概念モデルを用いている。2つの最も一般的なファイルモデルに使われている基準は、構造と変更可能性である。これらの基準に基づくファイルモデルを以下に述べる。

9.3.1 非構造化ファイルと構造化ファイル

最も簡素なモデルでは、ファイルは構造化されていないデータの並びである。このモデルでは、ファイルサーバによって知られている部分構造はなく、ファイルシステムの各ファイルの内容はファイルサーバにとって解釈されないバイトの並びとして見えている。オペレーティングシステムは、ファイルに格納されている情報に興味はない。したがって、ファイルに格納されているデータの意味や構造の解釈はすべて応用プログラムで行われる。UNIX や MS-DOS はこのファイルモデルを使っている。

現在ではまれにしか使われていないもう1つのファイルモデルは構造化されたファイルモデルである。このモデルでは、ファイルはファイルサーバに整列されたレコードの並びとして見えている。同じファイルシステムの異なるファイルのレコードは、異なるサイズとなってもよい。したがって、多くのタイプのファイルがファイルシステムに存在し、それぞれは異なる特性を持っている。このモデルでは、レコードがファイルデータのアクセスの最も小さな単位であり、ファイルシステムはレコードの集まりに対して読み込みや書き込みが実行される。

構造化ファイルは、また更に2つのタイプに分けられる。つまり、索引なしレコード **(nonindexed records)** と索引付きレコード **(indexed records)** である。前者のモデルでは、ファイルレコードはファイル内の位置を指定することでアクセスされる。例えば、ファイルの先頭から5番目のレコード、あるいはファイルの最後から2番目のレコードという指定である。後者のモデルでは、レコードには1つ以上のキーフィールドがあり、キーフィールドの値を指定することでレコードを特定することができる。索引付きレコードが使えるファイルシステムでは、ファイルはB-ツリーあるいは他の適切なデータ構造、あるいはハッシュテーブルを使うことでレコードの位置を素早く特定するように管理される。

最も最新のオペレーティングシステムは、非構造化ファイルモデルを使って

いる。これは、主に異なるアプリケーションによるファイルの共有が構造化ファイルモデルに比べて非構造化ファイルモデルのほうが容易であるという理由による。非構造化モデルではファイルは構造を持たないため、様々なアプリケーションがファイルの内容を様々な方法で解釈することができる。

データ項目に加えて、ファイルは通常**属性 (attributes)** も持つ。ファイルの属性とは、そのファイルの情報を記述したものである。各属性には、名前の値がある。例えば、ファイルの典型的な属性には所有者、サイズ、アクセスパーミッション、作成日時、最後の更新日時、そして最後のアクセス日時が含まれる。利用者は、ファイルシステムが提供するプリミティブを使っていくつかの属性値を変更することができる。しかし、利用者は任意の属性値を読むことはできても、すべての属性値を更新することはできない。例えば、利用者はアクセスパーミッションの属性値は更新できるかもしれないが、サイズや作成日時の属性値を更新することはできない。ファイルに関する属性のタイプは、ファイルシステムによって通常固定されている。しかし、ファイルシステムがサポートする属性に加えて、利用者が定義する属性を作ったり操作したりする柔軟性を提供するようにファイルシステムを設計することもできる。

ファイルの属性は、通常ディレクトリサービスによって管理され用いられる。それは、ファイル自身よりも様々なアクセス制御に従う必要があるからである。ファイル属性はディレクトリサービスによって管理され、用いられるが、それらはファイル名とともにディレクトリに格納されるのではなく対応するファイルとともに格納されることに注意する必要がある。これは、主に多くのディレクトリシステムがファイルを1つ以上の名前で参照できるようにしているためである。

9.3.2　変更可能ファイルと不変ファイル

変更可能性の基準により、ファイルは2つのタイプに分けられる。変更可能なタイプと不変のタイプである。ほとんどの既存のオペレーティングシステムは変更可能ファイルモデルを使っている。このモデルでは、ファイルへの更新は、新しい内容を作成するために古い内容にファイルを上書きすることが行われる。つまり、ファイルは各更新操作により変更される単一の格納された並びとして表現される。他方、例えば Cedar ファイルシステム (CFS)[Gifford et al. 1988] のような、少し新しいファイルシステムでは、不変ファイルモデルが使われている。このモデルでは、ファイルは削除される以外は一度作成される

と変更されることがない。ファイル版方式が通常ファイルの更新に使われ、そして各ファイルは不変である版の履歴によって表現される。つまり、同じファイルを更新するのではなく、ファイルに変更が行われるたびにファイルの新しい版が作成され、古い版は変更しないまま維持される。現実には、使用する記憶空間は、全ファイルを再度作り直すのではなく、古い版と新しい版との違いがあるレコードのみを保持することによって削減される。

Gifford et al. [1988] では、不変ファイルのみによる共有は一貫性のある共有をサポートすることが容易になることを強調している。この特徴から、不変ファイルモデルを使った分散システムのファイルキャッシュや複製のサポートが非常に容易になる。それは、複数のファイルのコピーの一貫性を管理することに関する問題がすべてなくなるからである。しかし、ファイルの複数の版維持する必要性から、不変ファイルモデルは 2 つの潜在的な問題に直面している。使用するディスク空間の増加とディスク確保動作の増加である。ディスク空間がすぐに一杯になることを避けるために、幾つかの機構が通常使われている。例えば、CFS[Gifford et al. 1988] は、keep パラメータがファイルの最新の版を維持する数として使われている。最新の版を追跡するために、各ファイル名は版番号が添え字として付けられている。例えば、“demo.abc!6” はベース名 demo.abc を持つファイルの第 6 版を示す。したがって、もしこのファイル keep パラメータが 1 であれば、新しいファイルの版を作成することは demo.abc!6 が削除され、そのディスク空間は新しいファイル demo.abc!7 のために再利用される。keep パラメータが 1 よりも大きい値となっている場合、つまり複数の版のファイルが存在するとき、利用者はその完全な名前を指定することで特定の版を使うことができる。しかし、ファイルの版番号を指定しない場合、デフォルトで CFS は delete などの幾つかの操作については最小の版番号を使い、open などの他の操作では最大の版番号を使う。

9.4 ファイルアクセスモデル

ファイルをアクセスするためのクライアントの要求方法は、ファイルシステムによって使われているファイルアクセスモデルに依存して提供される。分散ファイルシステムのファイルアクセスモデルは、2 つの要因に依存する。それは、遠隔のファイルをアクセスする方法とデータアクセスの単位である。

9.4.1　遠隔ファイルのアクセス

分散ファイルシステムは、アクセスされるファイルが遠隔ファイルである場合、クライアントのアクセス要求は以下のモデルのうちの 1 つを使ってサービスされる。

1. **遠隔サービスモデル (remote service model)**: このモデルでは、クライアントの要求の処理はサーバのノードで行われる。つまり、クライアントのファイルアクセスの要求はサーバに配送され、サーバマシンはアクセス要求を実行し、最後に結果がクライアントに返される。クライアントからのアクセス要求とサーバのクライアントへの応答はネットワークを通してメッセージとして転送される。メッセージ毎のデータをまとめたり通信するためのオーバヘッドは大きなものになる可能性がある。したがって、もし遠隔サービスモデルを使うときは、ファイルサーバインタフェースと通信プロトコルを、個々の要求を満足するための交換すべきメッセージ数とともにメッセージを生成するオーバヘッドを最小にするように注意深く設計する必要がある。

2. **データキャッシュモデル (data-caching model)**: 遠隔サービスモデルでは、各遠隔ファイルアクセス要求はネットワークトラフィックを発生する。データキャッシュモデルは、ファイルアクセスの局所性の利点を活かして、ネットワークトラフィックの量を減らすことを目的としている。このモデルでは、クライアントのアクセス要求を満足させるために必要とされるデータがローカルに存在しない場合、それをサーバのノードからクライアントのノードにコピーしてそこにキャッシュする。クライアントの要求は、キャッシュされたデータを使いクライアントのノード内で実行される。最新のアクセスされたデータはキャッシュの中にしばらく保持されるため、繰り返しのアクセスがローカルに実行される。**最新アクセス最小 (least recently used: LRU)** といった置き換え方策がキャッシュのサイズを有限に保つのに使われる。

遠隔アクセスモデルと違って、このモデルはネットワークトラフィックを大きく削減する。しかし、このモデルではある特定のファイルが異なるキャッシュ上で同時に利用可能となる可能性がある。したがって、書き込み操作は、適当な共有セマンティクスに依存して、そのデータのローカルにキャッシュされたコピーに変更を加える以外に、その変更をサーバノードの元のファイルとその

データを持っている他のすべてのキャッシュに行わなければならず、かなりのオーバヘッドを生じることがある。キャッシュされたデータを元のファイルの内容と一貫性を維持する問題は、**キャッシュ一貫性問題 (cache consistency problem)** と呼ばれる。この問題を扱う方法は、この章の後で述べる。

遠隔サービスモデルに比べて、データキャッシュモデルは性能の向上とシステム拡張性の向上の可能性を提供している。それは、ネットワークトラフィックを軽減し、ネットワークの競合を減らし、そしてファイルサーバの競合を減らしているからである。したがって、ほとんどの既存のファイルシステムは何らかの形式のキャッシュを実装している。実際には、多くの実装が遠隔サービスとデータキャッシュモデルとのハイブリッドと考えられる。例えば、LOCUS[Popek and Walker 1985] やネットワークファイルシステム (NFS)[Sandberg et al. 1985] は遠隔サービスモデルを使っているが性能の向上のためにキャッシュが追加されている。他方、Sprite[Nelson et al. 1988] はデータキャッシュモデルを使っているが、ある環境では遠隔サービス方式をも使っている。

9.4.2 データ転送の単位

データキャッシュモデルを使っているファイルシステムにおいて、重要な設計項目はデータ転送の単位を決定することである。データ転送の単位は、ファイルデータの断片 (あるいは複数の断片) を指し、1つの読み込みあるいは書き込み操作の結果としてクライアントに向かってそしてクライアントから転送される。この要因に基づく4つの共通に使われているデータ転送モデルを以下に示す。

1. **ファイルレベル転送モデル (file-level transfer model)**: このモデルでは、ある操作で、ファイルデータがクライアントとサーバ間を双方向にネットワークを通して転送することが必要となったとき、ファイル全体が移動するモデルである。

この概念的な簡潔さに加えて、このモデルは幾つかの利点を持っている [Satyanarayanan et al. 1985]。第一は、単一の要求の応答にファイル全体を送信することは、幾つかの要求に対してページ単位で転送するのに比べてより効率的である。なぜなら、ネットワークプロトコルのオーバヘッドが1度ですむからである。第二に、これはより拡張性がある。なぜなら、ファイルサーバにより少ないアクセスですみ、結果的にサーバの負荷とネットワークトラフィッ

クを軽減するからである。第三に、サーバのディスクアクセスルーチンは、ランダムなディスクブロックを要求することに比べて常にファイル全体を要求されることが分かっていればよりよい最適化が可能である。第四に、クライアントサイトにファイル全体が一度キャッシュされると、サーバやネットワーク障害を免れることができる。そのため、このモデルは本質的にある程度の回復性をも提供する。最後に、このモデルは異機種ワークステーションのサポートの作業を簡単にする。その理由は、ファイル全体を一度にサーバワークステーションのファイルシステムに互換性のある形式からクライアントのワークステーションのファイルシステムに互換性のある形式へ変換すること及びその逆の変換はより容易であるからである。

他方、このモデルの主な欠点は、クライアントのノードに必要とするファイルのすべてを格納するための十分な記憶空間が必要となることである。したがって、この方式は非常に大きなファイルについては使えない。特に、クライアントがディスクレスワークステーションで動作しているときには使えない。クライアントのワークステーションがディスクレスでない場合でも、ローカルディスクの容量より大きいファイルについては、全体をアクセスできない。更に、もしファイルにおける小さな断片のみしか必要としない場合、ファイル全体を移動することは無駄がある。

Amoeba[Mullender and Tanenbaum 1984]、CFS[Gifford et al. 1988]、そして Andrew File System(AFS-2)[Satyanarayanan 1990b] はファイルレベル転送モデルを使っている分散システムの例である。AFS-3 は AFS-2 の次版であるが、ファイル全体ではなく大きな塊 (64 キロバイト) でファイルをキャッシュすることも可能としている。この機能は、大きすぎてローカルディスクにおさまらないファイルをアクセス可能とするために AFS に後で組み込まれた。

2. ブロックレベル転送モデル (**block-level transfer model**): このモデルでは、ネットワークを通してクライアントとサーバ間で転送されるデータはファイルブロックの単位である。ファイルブロックは、ファイルの隣接する一部であり、通常長さが固定されている。ブロックサイズが仮想メモリのページサイズと等しいファイルシステムについては、このモデルはページレベル転送モデルとも呼ばれる。

このモデルの利点は、クライアントノードが大きなディスク空間を必要としないことである。この方法では、ファイルデータの小さな一部のみしか必要としない場合、ファイル全体をコピーする必要性をもなくしている。した

がって、このモデルではディスクレスワークステーションでも利用することができる。このモデルは、自身の2次記憶装置を持たないクライアントノードに大きな仮想的な記憶を提供する。しかし、ファイル全体をアクセスする場合、複数のサーバ要求がこのモデルでは必要であり、結果的により多くのネットワークトラフィックとプロトコルオーバヘッドを生じる。したがって、アクセス要求がほとんどのファイルが全体を転送しなければならないような場合、このモデルはファイルレベル転送モデルと比べて低い性能となる。Apollo Domain File System[Leach et al. 1983]、Sun Microsystems の NFS[Sandberg 1987]、LOCUS[Popek and Walker 1985] そして Sprite[Nelson et al. 1988] はブロックレベル転送モデルを使っている分散システムの例である。

3. **バイトレベル転送モデル (byte-level transfer model)**: このモデルでは、サーバとクライアント間でネットワークを通して転送されるファイルデータはバイト単位である。このモデルは、最大の自由度を提供する。なぜなら、この方式はファイル内のオフセットと長さを指定することによって、任意の連続するファイルの一部を格納したり取り出したりすることができるからである。このモデルの主な欠点は、様々なアクセス要求に対する可変長のデータのために、キャッシュを管理することが難しいことである。Cambridge File Server[Dion 1980, Mitchell and Dion 1982, Needham and Herbert 1982] がこの方式を使っている。

4. **レコードレベル転送モデル (record-level transfer model)**: 上で述べてきた3つのファイルデータ転送モデルは、非構造化ファイルモデルで共通に使われている方法である。レコードレベル転送モデルは、ファイルの内容がレコードの形式で構造化されているファイルモデルに対して使われるのに適している。このモデルでは、クライアントとサーバ間でネットワークを通して転送されるデータはレコードの単位である。Research Storage System(RSS)[Gray 1978, Gray et al. 1981] は、構造化された索引付きファイルに対する複雑なアクセス方式を提供しているが、このレコードレベル転送モデルを使っている。

9.5 ファイル共有セマンティクス

共有されたファイルは、複数の利用者によって同時にアクセスされることがある。このような場合、どのファイルシステムにとっても重要な設計項目は、

利用者によって行われたファイルデータへの変更をいつ他の利用者に観測できるようにするかを明確に定義することである。これは、ファイルシステムによって採用されたファイル共有セマンティクスのタイプによって定義される。Levy と Silberschatz[1990] は以下のファイル共有のタイプを定義した。

1. **UNIX セマンティクス (UNIX semantics)**: このセマンティクスは、すべての操作に絶対時間の順序を強制するもので、ファイルに対する各読み込み操作はそのファイルに対する以前のすべての書き込み操作の結果が見えることが保証されている [図 9.1(a)]。特に、ある利用者によるオープンされたファイルへの書き込みは、同時にこのファイルをオープンしている他の利用者にすぐに見えるようになる。

この UNIX セマンティクスは、単一のプロセッサシステムのファイルシステムに広く実現されている。なぜなら、これが最も望ましいセマンティクスであり、読み込み／書き込み要求を逐次化するのが容易だからでもある。しかし、分散ファイルシステムにおいて UNIX セマンティクスを実現するのは容易ではない。このセマンティクスは、分散システムにおいては、クライアントノードにおけるキャッシュを禁止して、ただ 1 つのファイルサーバにのみ共有ファイルを管理させ、すべての読み込みと書き込みの要求を厳密に受け取った順番で実行することによって達成できると思うかもしれない。しかし、この方法を取ったとしても、ネットワーク遅延によって、様々なノードからのクライアントの要求は現実にその要求が作成された順番とは異なって到着し、サーバノードで処理されることがある [図 9.1(b)]。更に、すべてのファイルアクセスを単一のサーバで処理すること、クライアントノードでのキャッシュを禁止することは性能の低下、拡張性の低下、そして信頼性の低下を分散ファイルシステムに生じるために現実的ではない。したがって、通常分散ファイルシステムはより緩やかなファイル共有セマンティクスを実現している。正しい動作のために UNIX セマンティクスを必要とするアプリケーションは、その目的のためには特別な処理を (例えばロック) を用いるべきで、ファイルシステムによって提供される下位の共有セマンティクスに依存すべきではない。

2. **セッションセマンティクス (session semantics)**: このセマンティクスに対して、以下のファイルアクセスパターンが仮定されている。クライアントはファイルをオープンし、ファイルに対して連続する読み込み／書き込み操作を行って、ファイルの利用が終了すると最後にファイルをクローズする。セッ

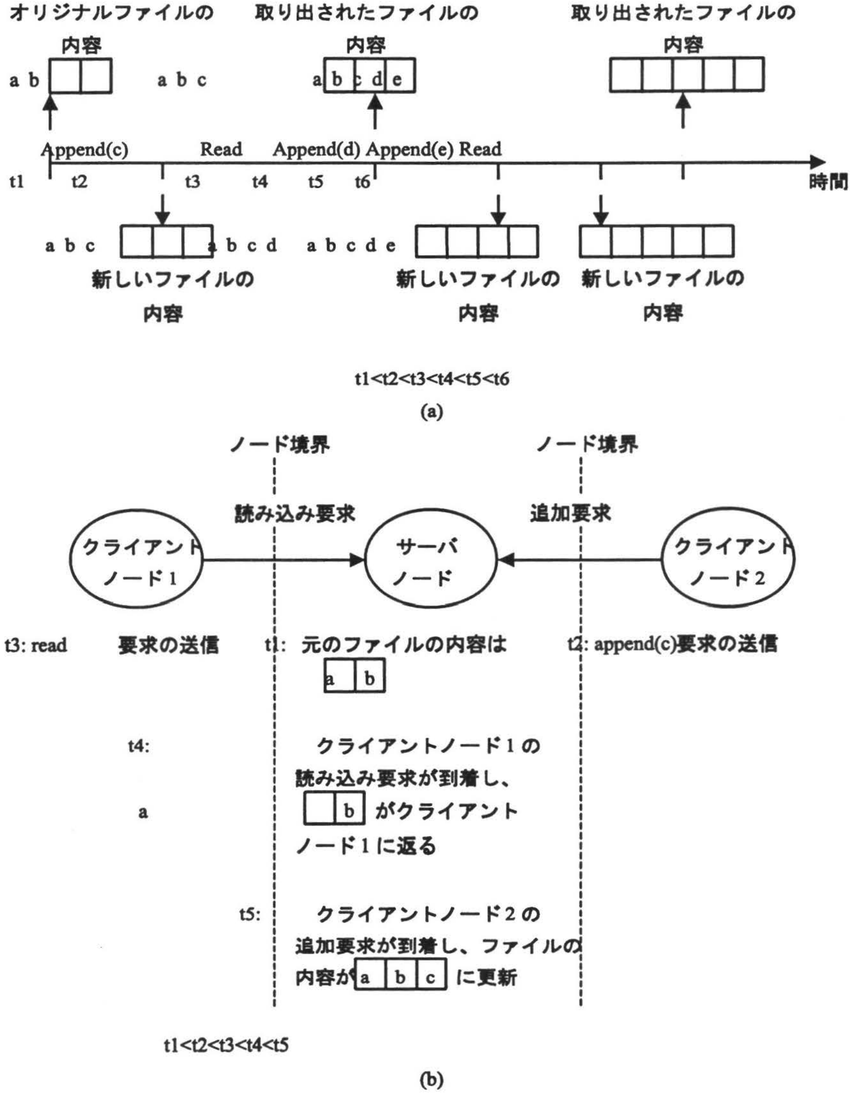


図 9.1 (a)UNIX ファイル共有セマンティクス, (b) 共有ファイルが単一のサーバによって管理されていてさえ分散ファイルシステムでは UNIX セマンティクスを達成することが難しいのはなぜかを示す例

ション (session) とは、オープンとクローズ操作の間に行われる一連のファイルアクセスである。セッションセマンティクスでは、セッション内で行われたファイルに対するすべての変更は、セッションを開いたそのクライアントプロセス (あるいはそのクライアントノードにあるすべてのプロセスについて同様) だけが初めに見える。そして、そのファイルを同時にオープンしている他の遠隔プロセスも見ることができる。一度セッションが閉じられると、ファイルに対する変更は後でセッションを開始した遠隔プロセスについてのみ見えるようになる。

セッションセマンティクスを使うと、複数のクライアントが同じファイルに対して並行に読み込みと書き込みが実行することができるようになる。この場合、各クライアントはそのファイルに対する各自のイメージを保持している。クライアントがそのセッションを閉じるとき、そのファイルの使用を継続している他の遠隔クライアントは、現実にファイルの古いコピーを使っていることになる。更に、セッションセマンティクスの使用は、複数のファイルセッションで各自が異なるファイルイメージを持っている場合、あるセッションがクローズした後で別のセッションがクローズしたときに、どのファイルイメージが最終のファイルイメージかという問題が生じる。セッションがクローズしたとき、ファイルイメージはサーバに戻される。そのため、最終のファイルイメージはだれが最後にクローズしたかに依存することになる。しかし、ネットワーク遅延により、異なるノードからのファイルイメージは、それが実際にクローズした順序と異なる順序でサーバに到着して処理される可能性がある。したがって、現実には最終的なファイルイメージは、動作中セッションの1つで保持されているイメージとなるという選択肢が、実装する上で容易であり、実際のイメージは規定しない。つまり、最終的なファイルイメージは非決定的である。

セッションセマンティクスは、ファイルレベル転送モデルを使っているファイルシステムにのみ使うことができる。なぜなら、セッションはファイルがオープンされているときにはファイル全体のイメージを読み込むことを仮定しているため、セッションセマンティクスとファイルの一部をキャッシュすることとの組み合わせは、複雑な方法となるからである。

3. 不変共有ファイルセマンティクス (Immutable shared-files semantics): このセマンティクスは、不変ファイルモデルの利用に基づいている。不変ファイルは、一度作成されると変更されることがない。このセマンティクスによれば、ファイルの作成者が一度それを共有可能とすると、そのファイルは不変ファ

イルとして扱われるため、もはや変更することはできない。ファイルへの変更は、そのファイルの新しい更新された版の作成として扱われる。ファイルの各版は、完全に新しいファイルとして扱われる。したがって、このセマンティクスはファイルが読み込み専用としてのみ共有されることを可能としている。この方式を使って、共有ファイルはもはや変更されないため、利用者によって加えられた変更をいつ他の利用者に見えるようにするかの問題はなくなる。

4. **トランザクション的セマンティクス (transaction-like semantics)**: このセマンティクスはトランザクション機構に基づいており、それは共有された可変データへの並行アクセスを制御する高レベルの機構である。**トランザクション (transaction)** は、begin_transaction と end_transaction の操作間に挟まれた操作の集合である。トランザクション機構は、共有データに対してトランザクションによって行われた部分的な変更が、そのトランザクションが終了する (その end_transaction が実行される) まで他の並行して実行されているトランザクションに対して見えないことを保証している。したがって、1つのファイルに対する複数の並行トランザクションにおいて、最終的なファイルの内容はすべてのトランザクションがある逐次的な順序で実行されたものと同じになる。Cambridge File Server[Needham and Herbert 1982] において、トランザクションの開始と終了はファイルの操作のオープンとクローズの内部に暗黙に組み込まれており、トランザクションは1つのファイルのみに関係することができる。したがって、そのシステムにおけるファイルセッションは実際にトランザクションである。

9.6 ファイルキャッシュ方式

ファイルキャッシュは、ファイル I/O 性能を向上させるために幾つかの時分割のファイルシステムで実現されてきた (例えば UNIX[McKusick et al. 1985])。これらのシステムにおけるファイルキャッシュの考えは、最近アクセスされたファイルデータを主記憶に保持するもので、同じ情報に対する繰り返しのアクセスは追加のディスク転送なしに扱うことができる。アクセスパターンの局所性のために、ファイルキャッシュはディスク転送を大きく減少させ、ファイルシステム全体の性能を向上させる。ファイルアクセスパターンの局所性についての性質は、適切なファイルキャッシュ方式を設計するために活用されている。性能の向上の他に、分散ファイルシステムのファイルキャッシュ方式はその拡

張性と信頼性にも寄与している。なぜなら、キャッシュを遠く離れたクライアントノードに置くこともできるからである。そのため、よく利用されている分散ファイルシステムでは今日いずれかの形式のファイルキャッシュを採用している。AT&T の遠隔ファイルシステム (Remote File System: RFS)[Rifkin et al. 1986] でさえ、現在はファイルキャッシュを使っている。

集中型ファイルシステムにおけるファイルキャッシュ方式の実現では、幾つかの重要な選択する必要がある。例えば、キャッシュされるデータの粒度 (大きいか、小さいか)、キャッシュサイズ (大きいか、小さいか、固定か、可変か) そして置き換え方策である。これらの設計項目についての良いまとめは、[Smith 1982] にある。これらの項目に加えて、分散ファイルシステムにおけるファイルキャッシュ方式は以下の重要な選択をもする必要がある。

1. キャッシュの位置
2. 変更の伝搬
3. キャッシュ検証

これらの3つの項目を以下に述べる。

9.6.1　キャッシュの位置

キャッシュの位置は、キャッシュデータが格納されている場所を示す。ファイルの元の位置はサーバのディスクとする。分散ファイルシステムでは3つの可能なキャッシュの位置がある (図 9.2)。

1. **サーバの主記憶:** キャッシュ方式を使わないとき、遠隔のクライアントがファイルをアクセスできるようになる前に、ファイルはサーバのディスクからサーバの主記憶に転送され、そしてネットワークを通してサーバの主記憶からクライアントの主記憶に転送される。したがって、全体のコストには1度のディスクアクセスと1度のネットワークアクセスが含まれている。サーバの主記憶にあるキャッシュはキャッシュヒット時にディスクアクセスコストを削減する。結果的に、キャッシュがないときに比べてかなりの性能向上につながる。

キャッシュをサーバの主記憶に配置する決定は、以下の1つ以上の理由によって行われる。この方式は実装が容易であり、クライアントにとって透過である。元々のファイルとキャッシュとの一貫性を維持することが、両者が同じノード

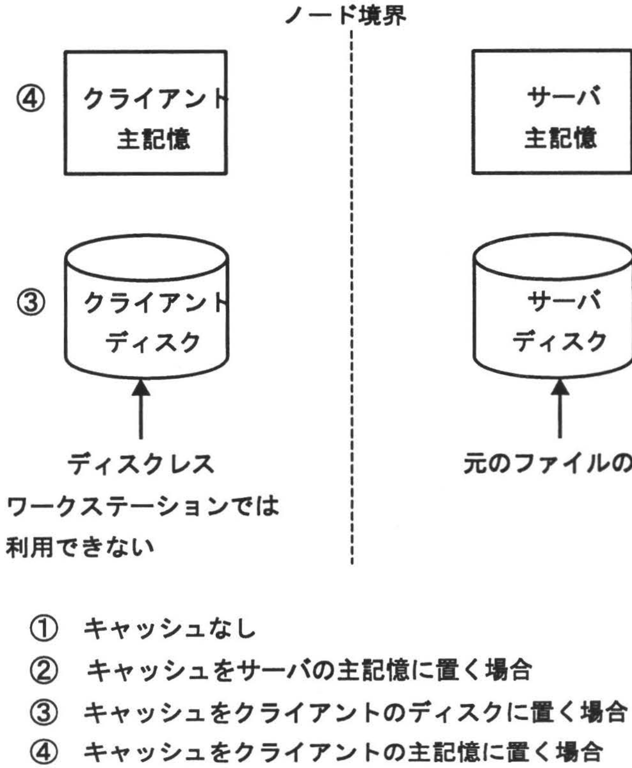


図 **9.2**　分散ファイルシステムのためのファイルキャッシュ方式における可能なキャッシュの配置

上にあるために容易である。更に、単一のサーバがキャッシュデータとファイルの両者を管理しているため、異なるクライアントによる複数のアクセスを容易に同期させて UNIX 的なファイル共有セマンティクスをサポートすることができる。

しかしながら、サーバの主記憶にキャッシュを持つことにより、遠隔のクライアントによるファイルアクセス操作毎にネットワークアクセスがあり、サーバによる処理が行われる。そのため、ネットワークアクセスコストの削減には寄与せず、分散ファイルシステムの拡張性と信頼性にも寄与しない。

2. **クライアントのディスク:** 2 番目の選択肢は、キャッシュをクライアントのディスクに持つことである。クライアントのディスクにキャッシュを持つことで、キャッシュヒット時にネットワークアクセスコストを削減することができる。ディスク上のキャッシュには幾つかの利点がある。その 1 つは信頼性である。キャッシュされたデータへの変更は、もしキャッシュが揮発性の記憶に管理されていたとすると、クラッシュによって失われてしまう。更に、もしキャッシュされたデータがクライアントのディスクに管理されているとき、キャッシュは回復時にもそこに置かれており、再度サーバから取り出してくる必要がない。2 つ目の利点は、大きな記憶容量である。主記憶によるキャッシュに比べて、ディスクキャッシュは非常に大きな記憶空間を持つ。したがって、より多くのデータがキャッシュ可能であり、結果的に大きなヒット率となる。更に、幾つかの分散ファイルシステムは、ファイル全体がキャッシュされるファイルレベルデータ転送モデルを使っている。これらのシステムでは、もしファイルが主記憶キャッシュでは大きすぎて入らないような場合、ファイルキャッシュの利点がそれによって失われる。そのため、ディスクキャッシュは、キャッシュされるべきファイルが利用可能なディスク空間よりも大きいことがないかぎりほとんどのファイルがキャッシュできるため、ファイル転送モデルを使っているこれらのシステムにとって現実的に利用可能である。3 つ目の利点は、接続なしの操作である。クライアントディスクキャッシュは、拡張性と信頼性に寄与する。なぜなら、アクセス要求のキャッシュヒットはサーバに接続することなくサービスをローカルに提供できるからである。

クライアントのディスクにキャッシュデータを持つことの主な欠点は、システムがディスクレスワークステーションをサポートしようとするときにこの方策はうまく動作しないということである。更に、このキャッシュの方策を使ったとき、キャッシュヒットが起こっても各アクセス要求毎にディスクアクセス

が必要である。したがって、アクセス時間はまだかなり大きい。サーバの主記憶のキャッシュは、キャッシュヒット時にディスクアクセスを削減したがネットワークアクセスは必要である。他方、クライアントのディスクキャッシュは、キャッシュヒット時にネットワークアクセスを削減するがディスクアクセスは必要である。したがって、キャッシュをサーバの主記憶とするかクライアントのディスクとするかについての選択を行う必要があるとき、前者のほうが幾らか速く、そして大抵はより簡単である [Tanenbaum 1995]。

3. クライアントの主記憶: 3 番めの選択しはクライアントの主記憶である。クライアントの主記憶にあるキャッシュは、ネットワークアクセスコストとディスクアクセスのコストの両者を削減することができる。したがって、この方法はキャッシュヒット時に最大の性能向上を得ることができる。この方法は、ワークステーションがディスクレスであってもよい。クライアントディスクキャッシュのように、クライアントの主記憶のキャッシュは拡張性と信頼性に寄与する。なぜなら、キャッシュヒット時にアクセス要求はサーバに接続する必要がなくローカルにアクセス要求に対してサービスされるからである。しかし、クライアントの主記憶でのキャッシュは、大きなキャッシュサイズやキャッシュされたデータの信頼性の向上が必要とされたときにクライアントディスクキャッシュよりも魅力がなくなる。

キャッシュの配置方策の相対的な利点が図 9.3 にまとめられている。まとめとして、主記憶キャッシュとディスクキャッシュは異なる機能性が強調されている。主記憶キャッシュはアクセス時間の削減が強調されているが、ディスクキャッシュはクライアントマシンの信頼性と自律性が強調されている。更に、サーバノードかクライアントノードにキャッシュを置く選択に直面したとき、後者が通常魅力がある。なぜなら、それは拡張性と信頼性の両方に寄与するからである。

9.6.2 変更の伝搬

キャッシュがクライアントノードに位置するファイルシステムでは、ファイルのデータは複数のノードに同時にキャッシュされることがある。このような状況では、これらすべてのノードのキャッシュは正確に同じファイルデータのコピーを持っているとき、そのキャッシュは一貫性がある (**consistent**) とい

キャッシュがない場合の遠隔アクセスのコスト
=1 度のディスクアクセス+1 度のネットワークアクセス

キャッシュの位置	キャッシュヒット時のアクセスコスト	利点
サーバの主記憶	1 度のネットワークアクセス	1. 実装が容易 2. クライアントに完全に透過 3. 元のファイルとキャッシュデータとの一貫性の管理が容易 4. UNIX ライクのファイル共有セマンティクスのサポートが容易
クライアントのディスク	1 度のディスクアクセス	1. クラッシュに対する信頼性 2. 大きな記憶容量 3. コネクション切断時の操作のサポートへの適合性 4. 拡張性と信頼性への寄与
クライアントの主記憶	ー	1. 最大の性能向上 2. ワークステーションがディスクレスでも可能 3. 拡張性と信頼性への寄与

図 **9.3**　3 つのキャッシュ配置方案の相対的な利点のまとめ

う。1つのクライアントによってファイルデータが変更され、他のノードにある対応したデータキャッシュが変更されないか無視されているとき、そのキャッシュは**一貫性がない (inconsistent)** 状態になりうる。

複数のクライアントノードにキャッシュされたデータを一貫性がある状態に保つことは、クライアントキャッシュを使う分散ファイルシステムにとって重要な設計項目である。この課題を扱うための様々な方法が提案されており、実装されてきた。これらの方法は、分散ファイルシステムについての以下に述べるキャッシュ設計項目に使われる方法に依存している。

1. キャッシュされたデータに対して加えられた変更をいつファイルサーバに伝搬させるか。
2. キャッシュデータをどのように検証するか。

変更の伝搬方式は以下に示す。また、キャッシュ検証方式は次の章で述べる。

分散ファイルシステムは以下に述べる変更伝搬方式のうちの1つを使っている。分散ファイルシステムによってサポートされるファイル共有セマンティクスは、使われる変更伝搬方式に大きく依存している。更に、変更伝搬方式はシステム性能と信頼性に大きな影響を持っている。

書き込みスルー方式

この方式では、キャッシュが更新されたとき、ファイルのマスタコピーを更新するために新しい値がすぐにサーバに送られる。この方法は、2つの主な利点がある。それは、高い信頼性とUNIX的なセマンティクスへの適合性である。各変更は、マスタコピーを持つサーバにすぐに送られるため、更新されたデータが失われる(クライアントがクラッシュした場合)リスクは非常に小さくなる。この方法の大きな欠点は、書き込み性能の低さである。なぜなら、各書き込みアクセスは、情報がサーバのマスタコピーに書き込まれるまで待つ必要があるからである。書き込みスルー方式を使ったとき、データキャッシュの利点は読み込みアクセスのみになる。それは、この遠隔サービス方式が基本的にすべての書き込みアクセスで使われるからである。したがって、この方式が適しているのは、読み込みの対書き込みの比率が高い場合に利用するときのみである。

遅延書き込み方式

書き込みスルー方式は読み込みを救うが、書き込みに関するネットワーク

ラフィックを削減することはない。したがって、書き込みのネットワークトラフィックを減らすために、幾つかのシステムでは遅延書き込み方式を利用している。この方式では、キャッシュが変更されると、新しい値はキャッシュのみに書き込まれる。そして、クライアントはキャッシュが更新されたことのみを通知する。しばらくした後で、1 つのファイルについてのすべての更新されたキャッシュが集められ、サーバに同時に送られる。

いつ変更がファイルサーバに送られるかによって、遅延書き込み方策は異なるタイプになる。3 つの良く使われる方法を以下に示す。

1. **キャッシュからの書き戻し時の書き込み**: この方法では、キャッシュの変更データは、キャッシュ置き換え方策がクライアントのキャッシュからキャッシュの書き戻しを決定したときにサーバに送られる。この方法は良い性能をもたらすが、しかしそれがサーバに書き込まれる前に、長い時間に渡ってデータがクライアントのキャッシュに存在している [Ousterhout et al. 1985]。このようなデータは、信頼性の問題を受けやすい。

2. **周期的書き込み**: この方法では、キャッシュは規定の時間間隔で周期的に調べられ、そして前回の調査から変更されたキャッシュデータのすべてをサーバに送る。Sprite[Nelson et al. 1988] はこの方法で 30 秒の時間間隔を使っている。

3. **クローズ時の書き込み**: この方法では、クライアントによるキャッシュデータによって加えられた変更は、クライアントによって対応するファイルがクローズされたときにサーバに送られる。クローズ時書き込みはセッションセマンティクスに完全に整合している。しかし、この方式は非常に短い期間しかオープンされなかったり、ほとんど変更されないファイルについては、ネットワークトラフィックを大きく削減することはない。更に、すべての変更データがクローズ操作の完了前にサーバに書き込まれる必要があるため、クローズ操作に長い時間がかかる。したがって、この方策はファイルが長時間オープンされ、頻繁に変更される場合に限って用いられるべきである。ITC ファイルシステム [Satyanarayanan et al. 1985] はクローズ時書き込み方策を用いている。

遅延書き込み方策は、以下の理由によって書き込みアクセスの性能を改善するのに役立つ。

1. 書き込みアクセスは、新しい値が書き込みを実行しているクライアントのキャッシュにのみ書き込まれるために、より迅速に完了する。

2. 変更されたデータは、それがサーバに送られる前に削除されることもある。例えば、多くのプログラムは一時ファイルを作成し、それを使って、それが作られたあとにすぐに削除される。このような場合、変更はすべてサーバに送る必要はなく、結果的に大きな性能向上につながる。

3. ファイルの更新をまとめてサーバに送ることは、それぞれを別々に送るよりも効率的である。

しかし、遅延書き込み方式は信頼性問題の影響を受ける。なぜなら、もしクライアントがクラッシュすると、変更がクライアントのキャッシュからサーバにまだ送られていないものが失われてしまう場合があるからである。この方法も別の欠点は、サーバに変更の伝搬が遅れるということは、曖昧なファイル共有セマンティクスを生じる。なぜなら、他のプロセスがファイルを読むと、タイミングに依存して得られる結果が異なるからである。

9.6.3 キャッシュ検証方式

ファイルデータは同時に複数のノードのキャッシュに存在することがある。変更の伝搬方策は、サーバノードにあるファイルのマスタコピーがいつキャッシュの変更に基づいて更新されるかを指定しているだけである。それは、他のノードのキャッシュに存在するファイルデータをいつ更新するかについては何も言っていない。明らかに、クライアントのキャッシュは、他のクライアントが対応するデータを更新すると古いものになってしまう。したがって、クライアントノードのキャッシュに存在しているデータがマスタコピーと一貫性があるかどうかを検証することが必要となる。もし一貫性がなければキャッシュされたデータは無効化され、データの更新された版をサーバから取り出してくる必要がある。キャッシュされたデータを検証する方法は基本的に 2 つある。それは、クライアント主導方式とサーバ主導方式である [Levy and Silberschatz 1990]。これらを以下に述べる。

クライアント主導方式

この方式では、クライアントはサーバに問い合わせて、そのローカルのキャッシュデータがマスターコピーと比べて一貫性があるかどうかを検査する。ファイル共有セマンティクスは、正当性検査の頻度に依存している。以下の方法のうちの1つが使われる。

1. **各アクセスの前の検査**: この方法は、キャッシュの主な目的を損なってしまう。なぜなら、サーバは各アクセス毎に問い合わせを受けることになるからである。しかし、UNIX 型のセマンティクスをサポートするには適している。

2. **定期的な検査**: この方法は、固定した時間間隔毎に検査が行われる。この方法の主な問題点は、ファイル共有のセマンティクスが曖昧になってしまうことである。なぜなら、アクセス操作が実行されるデータがタイミングに依存してしまうからである。

3. **ファイルオープン時の検査**: この方法では、クライアントのキャッシュは、クライアントが対応するファイルを使うためにオープンした時にのみ検査が行われる。この方法は、セッションセマンティクスをサポートするのに適している。分散ファイルシステムにおけるセッションセマンティクスの実装のための1つの方法は、ファイルレベル転送モデルにクローズ時書き込みによる変更伝搬方策とファイルオープン時検査によるキャッシュの検証方策の組み合わせである。

検証は、データのキャッシュされた版とサーバのマスタコピー版との最終変更の時間を比較することで行われる。もし、2つが同じなら、キャッシュデータは最新である。そうでなければ、キャッシュは古いものとなり、したがって現在の版をサーバから取り出すことになる。タイムスタンプの代わりに、版番号や検査合計が使われることもある。

サーバ主導方式

もし正当性の検証の頻度が高い場合、クライアント主導キャッシュ検証方式は非常に多くのネットワークトラフィックを発生し、サーバの多くの CPU 時間を消費することになる。この理由から、初期にクライアント主導方式を使っていた AFS(AFS-1 において) は、サーバ主導方式に切り替えた (AFS-3, AFS-3

において)[Satyanarayanan 1990b]。

この方法では、クライアントはファイルサーバにファイルをオープンするときに、ファイルが読み込みでオープンされたのか、書き込みでオープンされたのか、その両者かを通知する。ファイルサーバは、どのクライアントがどのファイルを何のモードでオープンしているかを記録しておく。この手順では、サーバは異なるクライアントによるファイルの利用モードをモニタしておき、それが潜在的な矛盾を生じる場合があることが検出されたときに反応する。潜在的な矛盾は、2つ以上のクライアントが対立するモードで1つのファイルをオープンしようとするときに発生する。例えば、もしファイルが読み込みのためにオープンされていたときに、他のクライアントが読み込みのためにオープンすることは何の問題もない。しかし、書き込みのためにオープンすることはできない。同様に、もしファイルが書き込みのためにオープンされていれば、新しいクライアントは何のモードでもオープンすることはできない。クライアントがファイルをクローズするとき、ファイルに加えたすべての変更とともにサーバに通告する。そのような通告を受けると、サーバはクライアントがどのファイルをどのモードでオープンしていたかの記録を更新する。

新しいクライアントが既にオープンされているファイルに対してオープンの要求をし、そしてそのオープンモードが既にオープンされているモードと対立する場合、サーバは要求を拒否するか、要求を待ち行列に入れるか、又はキャッシュを利用不可能にして、すべてのこのファイルのキャッシュを持つクライアントに対してキャッシュからこのファイルを除くように依頼することで、その特別のファイルを遠隔サービスモードに切り替えるように設計することになる。キャッシュを利用不可能にする方法は、Sprite[Nelson et al. 1988] によって利用されている。

上で述べたサーバ主導方式は非常に効率的であるが、以下のような問題もある。

1. クライアントから起動されたサービス要求にサーバが単に応答するだけという、従来のクライアント-サーバモデルに違反している。このことは、クライアントとサーバのプログラムを例外的で複雑にする。

2. ファイルサーバがステートフルになることが必要である。後で述べるように、ステートフルのファイルサーバは障害が発生した場合にステートレスのファイルサーバに比べて異なる欠点が生じる。

3. オープン時検査、クライアント主導キャッシュ検証方式がサーバ主導方式とともに使われなければならない。例えば、クライアントはファイルをオープンし、それをキャッシュし、そしてそれを使った後でクローズする。それを再度使用するためにオープンすると、キャッシュの内容は検証されなければならない。なぜなら、他のクライアントが続いてファイルをオープンして変更してそしてクローズした可能性があるからである。

もう 1 つのサーバ主導の方式、それは AFS における**コールバック方策 (callback policy)** として知られているが [Satyanarayanan 1990b]、キャッシュはサーバによって通知されたものがなければ正しいと仮定する。この方法では、オープンされたファイルに関するクライアントのアクセスモードをすべて記録する代わりに、サーバはだれがファイル (あるいはその一部) をキャッシュしたかを記録する。この記録は、基本的にはファイル毎に管理され、サーバによって管理された各ファイルに対応して、サーバはクライアントがそのファイルを現在オープンしているかどうかによらず、ファイルのデータを自分のキャッシュ内に持っているクライアントのリストを管理する。サーバは、すべてのクライアントに対してキャッシュしたファイルが他のクライアントによって変更される前に通知することを約束している。

AFS においては、セッションセマンティクスを実現しているが、サーバが変更されたファイルのクローズの要求を受け取ったとき、サーバはそのファイルデータを持っているすべてのクライアントに対して、キャッシュデータを消去して無効とすることを通知する。このファイルをオープンして持っているクライアントは、現在のセッションが完了した時点でそのコピーを消去する。他のクライアントはそのコピーをすぐに削除する。この方法では、サーバはキャッシュされたファイルが既にオープンされているかどうかを知る必要がない。書き込みセッションがクローズされたことについてのみ知る必要があるだけである。結果的に、サーバはローカルにキャッシュされたファイルのオープン時の検証要求を受ける必要はない。もし、クライアントマシンがクラッシュしたとき、すべてのそのローカルファイルは、一貫性がなくなっているとみなされる。したがって、キャッシュからの回復について、ローカルディスクにキャッシュされた各ファイルについてキャッシュ検証要求が生成される。

9.7 ファイル複製

高可用性は良い分散ファイルシステムにとって望ましい性質であり、ファイルの複製はファイルの可用性を向上させる主要な機構である。**複製されたファイル (replicated file)** とは、複数のコピーを持つファイルで、各コピーは別のファイルサーバに存在しているものである。複製されたファイルからなるコピー集合における各コピーは、ファイルの**複製 (replica)** と呼ぶ。

9.7.1 複製とキャッシュの違い

複製は、しばしばキャッシュと誤解される。それは、恐らく両者とも複数のコピーを扱っているからである。しかし、2つの概念は以下のような基本的な違いがある。

1. 複製は、サーバに対応し、キャッシュされたコピーは通常はクライアントに対応する。

2. キャッシュされたコピーが存在する理由は、主にファイルアクセスパターンの局所性に主に依存している。一方、複製の存在する理由は、通常可用性や性能の要求に基づくものである。

3. キャッシュされたコピーと比較して、複製はより永続的であり、広く知られ、安全で、広く利用可能で、完全であり、そして正確である。

4. キャッシュされたコピーは複製に比べて臨時的である。複製に関して周期的な再検証をするような場合を除けば、キャッシュされたコピーの方が、有効になることはない。

Satyanarayanan[1992] は、複製されたコピーとキャッシュされたコピーを、first-class replica と second-class replica と呼んで区別した。

9.7.2 複製の利点

分散システムのデータの複製には、以下の潜在的な利点がある。

1. **可用性の向上:** 複製の最も重要な利点の1つが、ネットワークにおける障害を優美にマスクして耐性をもつということである。特に、障害時においても、利用者に操作性と可用性を残すことができる。独立した障害モードを持つサーバ上に重要なデータを複製することにより、1つのデータのコピーがアクセスできる確率が増加する。したがって、複製されたデータの代替のコピーは主コピーが利用できないときに使うことができる。

2. **信頼性の向上:** 多くのアプリケーションは、ファイルに格納されているそれらのデータの高い信頼性を要求する。複製は、このようなアプリケーションに非常に効果がある。なぜなら、それらのファイルの複数のコピーの存在を可能としているからである。システムの冗長な情報の存在によって、破局的な障害 (例えば、記憶装置のデータの永久的な消滅) からの回復が可能になる。

3. **応答時間の改善:** 複製は、応答時間を改善することにも役に立つ。それは、複製がデータをローカルにアクセスすることや、もとのコピーをアクセスするよりもアクセス時間が小さいノードからのアクセスを可能することによる。アクセス時間の差は、ネットワークトポロジやノードの負荷によっても生じる。

4. **ネットワークトラフィックの削減:** もしファイルの複製がクライアントのノードにあるファイルサーバで利用可能であれば、クライアントのアクセス要求がローカルで実行され、結果的にネットワークトラフィックを削減する。

5. **システムスループットの改善:** 複製は、同じファイルへのクライアントからのアクセス要求を異なるサーバで並行に実行することができ、結果的にスループットが向上する。

6. **拡張性の向上:** 1つのファイルを共有している利用者の数が増大するにつれて、1つのファイルサーバによって、ファイルに対するすべてのアクセス要求を実行することは、ファイルサーバの過負荷による性能の悪化を招く。複数のサーバ上にファイルを複製することによって、同じ要求は作業負荷の分散により更に効率的に複数のサーバで実行される。この結果は、拡張性の向上につながる。

7. **自律的な操作:** クライアントに対するサービスとしてファイルの複製を提

供している分散システムでは、制限された時間間隔だけクライアントに要求されたすべてのファイルをクライアントノードに近いファイルサーバに複製することができる。これは、クライアントマシンの一時的な自律的操作を可能とする。この機能を持つ分散システムは、取り外し可能で携帯可能なマシンをサポートする。

9.7.3 複製透過

ファイル複製において、透過性は重要な項目である。複製されたファイルサービスは、複製されていないファイルサービスのように機能しなければならないが、性能と信頼性の向上を示す必要がある。つまり、ファイルの複製は利用者に透過に設計されるため、複製されたファイルの複数のコピーが利用者に単一の論理的なファイルのように見える。このためには、読み込み、書き込み、そして他のファイル操作は、それが複製されていないファイルでも複製されているファイルでも同じクライアントインタフェースを持つべきである。透過性に関連する2つの重要な課題が、名前付けと複製制御である。

複製の名前付け

オブジェクトの複製をサポートしているシステムでは、1つのオブジェクトの異なる複製は同じ識別子をつけるべきか、異なる識別子をつけるべきかは基本的な問題である。明らかに、複製透過の要求によればひとつのオブジェクトに対するすべての複製は同じ識別子を割り当てると考える。1つのオブジェクトのすべての複製に1つの識別子を割り当てることは、不変オブジェクトでは妥当な考えである。なぜなら、カーネルはこのタイプのオブジェクトを容易にサポートできるからである。カーネルによって見えるすべてのコピーを使うことができる。それは、すべてのコピーが不変であり、同一であるからである。つまり与えられた識別子には、1つの論理的なオブジェクトしか存在しないからである。しかし、変更可能なオブジェクトでは、その時点でのインスタンスとして、複製されたオブジェクトの異なるコピーが同じ (一貫性がある) ではないことがある。この場合、オブジェクトのすべての複製に同じ識別子が使われていると、カーネルはどちらの複製が最新のものかを決定することができない。したがって、変更可能なオブジェクトの様々な複製の一貫性制御と管理がカーネルの外で実行されなければならない。そのため、利用者が与えた識別子を変更可能なオブジェクトの適切な複製に対応づけることは名前システムの責

任となる。さらに、もし、すべての複製が一貫性があれば、対応づけ機構はすべての複製の位置を提供し、利用者のノードからの相対的な距離を求める機構を提供する必要がある。

複製制御

もう1つの透過性の項目は、複製制御である。複製制御には、複製されたファイルの数と位置を決定することも含まれる。つまり、複製されたファイルのコピーを幾つ作るべきか、そのコピーをどのサーバに配置するべきかを決定する役目である。複製透過なシステムでは、複製制御は完全に自動的に、利用者透過な方法で行われる。しかし、ある環境では、これらの詳細を利用者に示して、複製プロセスを制御する柔軟な方法を提供することも必要となる。例えば、もし複製機能がワークステーションの自立的な操作をサポートするように提供されているとき、利用者には必要なファイルの複製をローカルノードに作れるような柔軟性が提供されているべきである。同様に、作業の性質のために、利用者が2つか3つの異なるノードに異なる時間に作業する場合、よく使うファイルをすべてのその利用者のワークステーションに複製しておくことが望ましい。

複製制御を利用者に透過にするかどうかに依存して、複製プロセスは2つのタイプに分けられる。

1. **明示的複製 (explicit replication)**:このタイプの複製では、利用者は複製プロセスの全体を制御する自由度を与えられる。つまり、プロセスがファイルを作るとき、そのファイルを置くべきサーバをしているする。そして、必要ならそのファイルの追加のコピーが利用者の明示的な要求によって他のサーバに作られる。利用者は、複製されたファイルの1つ以上の複製を削除する自由度を持っている。

2. **暗黙的／遅延複製 (implicit/lazy replication)**:このタイプでは、複製プロセスの全体はシステムによって自動的に制御され、利用者の知識は入る余地がない。つまり、プロセスがファイルを作成するときは、その位置に関する情報は何も提供しない。システムは自動的にファイルを配置するためのサーバを選択する。その後、システムによって使われるある複製方策によって、システムは自動的にファイルの複製を他のサーバ上に作成する。システムは、必要な複製が作成され存在するように、十分に知的である必要がある。また、必要がなくなれば無駄なコピーは自動的に削除できる必要がある。遅延複製は、通常サーバがある空き時間ができたときにバックグラウンドで実行されるもので

ある。

9.7.4 複数コピーの更新問題

ファイルシステムが同じ (論理的に) ファイルの複数のコピーを異なるサーバに置くことを可能にすれば、すぐにそれらを相互に一貫性のある状態に維持するための問題に直面する。つまり、同じファイルの異なるコピーに対して、独立の矛盾する更新が発生することを認めないようにする必要がある。実際、複製されたファイルが更新されるとき、コピー間の一貫性を維持することはファイルの複製をサポートするファイルシステムには大きな設計項目である。この問題を扱うために一般に用いられているいつかの方法を以下に述べる。

読み込み専用複製 (read-only replication)

この方法は、不変ファイルのみを複製を認める方法である。不変ファイルは読み込み専用でのみ使われるため、そして変更可能なファイルが複製されないため、複数コピーの更新問題は発生しない。読み込み頻度が高く一定時間に1度しか更新がない (数か月に1度など) ファイル、例えばシステムプログラムのオブジェクトコードが含まれるファイルなどは、不変ファイルとして扱いこの方法によって複製することができる。

任意読み込み全書き込みプロトコル (read-any-write-all protocol)

読み込み専用複製方式は、不変ファイルのみを複製を認めるということから非常に制限のある方法である。明らかに、変更可能なファイルの複製をサポートする複製方式が必要である。変更可能なファイルの複製を認める簡単な複数コピー更新プロトコルは、任意読み込み全書き込みプロトコルである。この方法では、複製されたファイルへの読み込み操作はファイルの任意のコピーを読み込むことによって行われ、書き込み操作はファイルのすべてのコピーに書き込むことで行われる。書き込み操作を実行するには、何らかのロックが使われる。つまり、どれかのコピーが更新される前には、すべてのコピーはロックされ、それから更新され、最後に書き込みを完了するためにロックが解放される。このプロトコルは、UNIX 的セマンティクスを実現するために適している。この方式では、書き込み操作の可用性が厳しく制限されている。なぜなら、その操作が完了するためにはすべてのコピーが利用可能でなければならないためである。しかし、読み込み操作は応答性が良い。なぜなら、最も近いコピーを

使うことができるからである。

利用可能コピープロトコル (available-copies protocol)

任意読み込み全書き込みプロトコルの主な問題は、書き込み操作において、もし複製されたファイルを持っているサーバのどれかがダウンしていれば、書き込み操作は実行できないということである。利用可能コピープロトコルは、この制約を緩和し、ファイルの複製を持っているサーバの幾つかがダウンしていても書き込み操作が実行できるようにしている。この方法では、読み込みは任意の利用可能なコピーから行われるが、書き込みはすべての利用可能なコピーに書き込むことで行われる。このプロトコルが正しく動作するための基本的な考えは、障害からサーバ復帰したとき、利用者の要求を受け付ける前に、他のサーバからコピーすることで最新の状態に移行する。失敗したサーバ (サイト) は高優先度状態管理ルーチンによって動的に検出され、システムから除外され、新しく回復したサイトは構成内に組み込まれる。このプロトコルは、任意読み込み全書き込みより良い可用性を提供するが、ネットワーク切断などによる通信障害が存在する場合の一貫性のなさを防ぐことはできない。この方法では、明確で検出可能なサイト故障は正確に扱うことができる。

主コピープロトコル (primary-copy protocol)

複数コピー更新問題を解決するもう 1 つの簡単な方法は、主コピープロトコルである。この方法は、各複製されたファイルに対して、1 つのコピーが主コピーとして選定され、すべての他のコピーは従コピーとする。読み込み操作は、主でも従でも任意のコピーを使って行われる。しかし、すべての書き込み操作は主コピーについてのみ直接行われる。従コピーを持っている各サーバは、主コピーを持っているサーバからの変更通知やコピーの更新通知の受信によって、そのコピーの更新を行う。

実現される一貫性セマンティクスは、いつ従コピーが更新されるかに依存している。例えば、UNIX 的セマンティクスに対しては、主コピーサーバが更新要求を受けたときに、それをすぐに他のすべての従コピーサーバに対して更新するように命令する。何らかのロックが用いられ、書き込み操作はすべてのコピーが更新されたときにのみ完了する。したがって、この場合、主コピー更新プロトコルは、任意読み込み全書き込みプロトコルの簡単なもう 1 つの実現となる。

もし、書き込み操作が主コピーが更新されるとすぐに完了するとき、より曖

味な一貫性セマンティクスが生じる。従コピーは、バックグラウンドで、あるいはサーバによって更新された版が要求されたときに、遅れて更新される。この方法では、すべての従コピーが結果的に更新され、一貫性のある状態に到達する。

定数に基づくプロトコル (quorum-based protocol)

任意読み込み全書き込みと利用可能コピープロトコルは、複製されたファイルのコピーが2つ以上の動作しているグループに分割される、ネットワーク分割問題を扱うことができない。更に、主コピー方式は主コピーを持っているサーバがダウンすると、書き込み操作が実行できなくなる点で、制限が多い。Gifford[1979a] は、ネットワーク分割問題を扱うことができ、読み込み操作よりも高価である場合の書き込み操作の利用可能性を向上させることができる、簡単な定数プロトコルを提案した。

定数に基づくプロトコルは、以下のように動作する。複製されたファイル F に全体で n 個のコピーがあるとする。ファイルを読むためには、F の最小で r 個のコピーを考慮する。この r 個のコピーの集合を**読み込み定数 (read quorum)** と呼ぶ。同様に、ファイルに書き込み操作をするためには、F の w 個のコピーに書き込まれなければならない。この w 個のコピーの集合を**書き込み定数 (write quorum)** と呼ぶ。r と w の値を選択するときの制約は、読み込みと書き込みの定数がコピーの総数 n よりも大きくなければならないということである ($r + w > n$)。この制限は、どの読み込み定数とどの書き込み定数にも0より多い数の重なりが存在していることを保証している。つまり、読み込みと書き込みのどの組み合わせでも、少なくとも1つの共通のコピーが存在しており、結果的に少なくとも1つの最新のコピーが読み込み／書き込み定数に存在しているということである。

定数プロトコルは、書き込み操作が複製されたファイルのすべてのコピーに実行されることがないため、幾つかのコピーは陳腐化する。そのため、定数の中で現在の (最新の) コピーを識別できることが必要となる。これは、通常各コピーの版番号属性を使って達成される。コピーの版番号は、コピーが変更されるたびに更新される。各コピーに付けられる新しい版番号は、現在のコピーについている版番号より1つ大きい番号となる。

読み込みは、以下のようにして行われる。

1. F の読み込み定数 (任意の r 個のコピー) を取り出す。

2. 取り出した r 個のコピーについて、最大の版番号を持つコピーを選択する。
3. 選択されたコピーに読み込み操作を実行する。

書き込みは、以下のようにして行われる。

1. F の書き込み定数 (任意の w 個のコピー) を取り出す。
2. 取り出した w 個のコピーについて、最大の版番号を持つコピーの版番号を得る。
3. 版番号を 1 増加させる。
4. 新しい値と新しい版番号を書き込み定数の w 個のコピーすべてに書き込む。

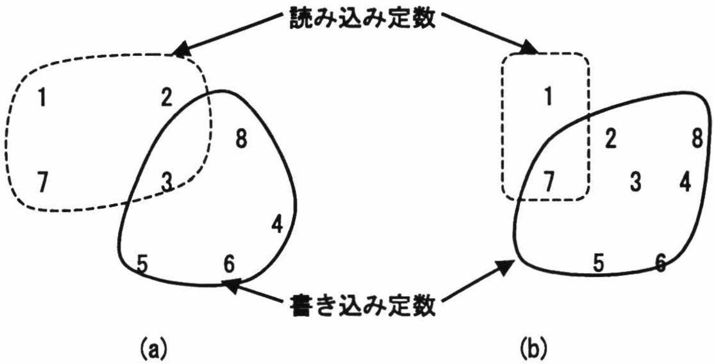


図 9.4　定数合意アルゴリズム.(a) $n = 8$、$r = 4$、$w = 5$、(b)$n = 8$、$r = 2$、$w = 7$

このプロトコルがどのように動作するかを明確化するため、図 9.4(a) の例を考える。複製された合計 8 個のファイル ($n = 8$) があり、読み込みと書き込みの定数をそれぞれ 4 と 5 とする ($r = 4, w = 5$)。したがって、条件 $r + w > n$ は満たされている。書き込み操作が、コピー 3, 4, 5, 6, そして 8 からなる書き込み定数に実行されたとする。すべてのこれらのコピーは新しい版を新しい版番号を得る。ここで、続く読み込み操作は 4 つのコピーからなる読み込み定数に実行される必要がある。それは、$r = 4$ だからである。明らかに、どの読み込み定数も前に実行された書き込み定数の少なくとも 1 つのコピーを含むことになる。図における読み込み定数では、コピー番号 3 が共通のコピーである。

読み込み定数に属しているコピーの版番号をみると、コピー番号 3 の版番号が最も大きい値であることが分かる。したがって、読み込み操作は、コピー番号 3 を使って実行される。$r = 2$ で $w = 7$ とした他の例が図 9.4(b) に示されている。

上に説明したプロトコルは、ごく一般的なもので、幾つかの特別なアルゴリズムが導出されている。そのなかの幾つかを以下に述べる。

1. **任意読み込み全書き込みプロトコル (read-any-write-all protocol)**:任意読み込み全書き込みプロトコルは、一般的な定数プロトコルを $r = 1$ と $w = n$ とした特別な場合である。このプロトコルは、読み込みの割合が書き込みに比べて大きい場合に使うことに適している。

2. **全読み込み任意書き込みプロトコル (read-all-write-any protocol)**:このプロトコルでは、$r = n$ で $w = 1$ である。このプロトコルは、書き込み比率が読み込みに比べて高い場合に使われる。

3. **過半数合意プロトコル (majority-Consensus protocol)**:このプロトコルでは、読み込み定数と書き込み定数とがほとんど等しいか同じである。例えば、もし $n = 11$ であれば、このプロトコルの可能な定数の割り当ては、$r = 6$、$w = 6$ となる。同様に、$n = 12$ のとき、可能な定数の割り当ては、$r = 6$、$w = 7$ である。このプロトコルは、一般に読み込みと書き込みの操作の割合がほぼ 1 であるような場合に使われる。

4. **重み付き投票による合意 (consensus with weighted voting)**:上で述べてきたすべての定数準拠のプロトコルでは、複製されたすべてのコピーは同じ重要度を与えられている。言い換えれば、すべてのコピーは 1 つの票を持っている。一般化された定数に基づくプロトコルは、各コピーにある票の数値を与えることによって、複製されたファイルの別々のコピーが持つ異なる重要度をモデル化することに使うこともできる。コピーごとの票は、性能や信頼性のなどに基づいて決定することができる。例えば、複製されたファイルの n 個の複製において、それらは別々のノードにあるとき、ノード A の複製が他の複製と比較して高い頻度でアクセスされているとする。このことは、ノード A のコピーが他の複製より多くの票を割り当てられているとモデル化することができる。

この方法では、ファイルを読むために r の票による読み込み定数が集められ、

ファイルを書くために w の票の定数が集められる。各コピーに割り当てられた票は同じではないため、読み込み／書き込みの定数のサイズは定数に集められたコピーに依存する。集められたコピーに割り当てられている票が相対的に大きい場合、定数に入るコピーの数は少なくなる。また、選択したコピーに割り当てられた票が小さい場合、定数に入るコピーの数は大きくなる。したがって、各読み込み定数と書き込み定数の間に 0 より大きい重なりがあるように保証するためには、r と w の値が、$r + w$ が票の総数 (v) よりも大きいように設定する必要がある ($r + w > v$)。ここで、v はすべてのコピーの票の合計である。

9.8　フォールトトレラント性

フォールトトレラント性は、分散ファイルシステムにおける重要な設計項目である。様々なタイプの障害により、このようなシステムに格納されているデータの一貫性を損なう。例えば、クラッシュイベントによってプロセッサは主記憶内の内容を失ってしまう。このような障害は、論理的には完了していても物理的には完了していないファイル操作を生じ、ファイルシステムに格納されているデータを一貫性のないものにする。同様に、要求が処理されている最中に、サーバやクライアントがクラッシュすることがあり、アクセスしているファイルの情報を失ってしまうことがある。これは、ファイルデータの一貫性に不明確な結果をもたらす。また、他の有害な環境上の事象、例えば一時的な障害 (電磁的な変動による) あるいはディスク記憶装置の故障においても、ファイルシステムに格納されているデータの損失や破壊が起こる。ディスク記憶装置の一部がデータを取り出せなくなっていることを “損傷”(“decayed”) という。

分散ファイルシステムの機能を障害に耐えるようにするのに直接関係する主なファイルの性質を以下に述べる [Levy and Silberschatz 1990]。

1. **可用性 (availability)**:ファイルの可用性とは、ファイルが利用可能になっている時間の断片を示している。可用性の性質は、ファイルの位置とそのクライアント (利用者) の位置に依存している。例えば、もしネットワークが通信リンクの故障で切断されたとき、ファイルが幾つかのノードのクライアントには利用可能かもしれないし、同時に他のノードには利用できないかもしれない。複製は、ファイルの可用性を高める 1 つの重要な機能である。

2. **堅固性 (robustness)**:ファイルの堅固性とは、ファイルが格納されている

記憶装置の故障や記憶媒体の損傷から生き残る能力を示している。冗長技術をよって実現されている記憶装置、例えば安定記憶は、堅固なファイルを格納するのによく使われる。堅固なファイルは障害のある要素が回復されるまで利用可能にならない。したがって、可用性とは異なり、堅固性はファイルの位置やそのクライアントの位置には依存しない。

3. **回復可能性 (recoverability)**:ファイルの回復可能性とは、ファイルへの操作が失敗したとき、あるいはクライアントによってアボートされたときに、初期の一貫性のある状態にロールバックする能力を指している。堅固なファイルが回復可能である必要はないし、その逆もない。トランザクション機構のような原子的更新技術が回復可能ファイルに使われる。

ファイル複製技術は前の章で既に述べてきた。原子的な更新に使われる原子トランザクション機構は次の章で述べる。以下では、安定記憶の技術と、分散ファイルシステムの耐障害性へのサービスパラダイムの効果について述べる。

9.8.1 安定記憶

クラッシュを防ぐ能力において、記憶装置は大きく以下の3つのタイプに分類される。

1. **揮発性記憶 (volatile storage)**:RAMのような記憶で、電源障害やマシンクラッシュに耐えることができない。つまり、揮発性記憶に格納されたデータは、電源障害やマシンクラッシュによって失われる。

2. **不揮発性記憶 (nonvolatile storage)**:ディスクのように、CPU障害には耐えるが、一時的なI/O障害や記憶媒体の損傷には耐えられない。十分信頼性はあるが、ディスクのような不揮発性記憶は複雑な障害モードを持ち、重要なデータを格納するには信頼性が十分でないことが明らかである。

3. **安定記憶 (stable storage)**:これは、一時的なI/O障害や記憶媒体の損傷にも耐えられる。これは、Lampson[1981]によって導入された記憶方式である。

安定記憶の基本的な考えは、安定な装置を実現するために2重化された記憶

装置を使うことと、どの時点でも 2 つの要素装置のうち 1 つのみが操作されることを保証することで、安定記憶の平均故障間隔 (MTBF) を向上させるものである。したがって、ディスクに基づく安定記憶システムは、独立に損傷する可能性のある既存の 1 組のディスク (ディスク 1 とディスク 2 と呼ぶ) からなる。ディスク 2 の各ブロックは、ディスク 1 の対応するブロックの正確なコピーである。ディスク記憶に影響する予測できない障害が起こるかもしれないが、同時に両者ディスクが損害を受けないことを保証するように効果的なフォールトトレラント機能が提供される。これは、2 つのディスクをアクセスする方法に制限を付けることによって達成されている。

従来のディスクにあるように、安定記憶に関連する 2 つの基本的な操作は読み込みと書き込みである。読み込み操作は、最初にディスク 1 に試みられる。もし、それが失敗すると、読み込みはディスク 2 に対して行われる。書き込み操作は、両方のディスクに行われるが、ディスク 2 への書き込みは、ディスク 1 への書き込みが正常に完了するまで開始しない。これは、ハードウェア障害によって 2 つのディスクが同時に被害を受けることを避けるためである。各ディスクに対する読み込みと書き込みの動作は、一時的なハードウェア障害に耐えるように再実行される。

更に、クラッシュが起こった後で、2 つのディスクに格納されたデータの内部的な一貫性を取り戻すクラッシュ回復動作が存在する。この回復動作は 2 つのディスクをブロック単位で比較していく。2 つの対応するブロックが異なるとき、正しくないデータを持っているブロックは、他のディスクの対応するブロックから再構築される。データブロックの正しさは、クラッシュが起こったタイミングに依存する。例えば、ディスク 1 が更新された後、ディスク 2 が更新される前、又はディスク 2 が更新される途中でシステムがクラッシュすると、ディスク 1 のデータブロックが正しいデータになる。他方、もしディスク 1 が更新されている途中でシステムがクラッシュすると、ディスク 2 が正しいデータとなる。もちろん、後者の場合は更新操作が最初から再実行されなければならない。

安定記憶システムは、元々誤りやすいディスクを使って、その障害の確率が無視できる程度の高信頼の仮想的な装置に変換している。安定記憶は高いフォールトトレラント性、例えば原子トランザクションを必要とするアプリケーションに適している。

9.8.2 フォールトトレラント性に対するサービスパラダイムの効果

サーバは、以下の2つのサービスパラダイムのうちの1つを使って実現される。それは、ステートフルとステートレスである。この2つのパラダイムは、クライアントサーバ関係の1つの側面によって識別される。それは、クライアントとサーバ間のサービスされた要求の履歴が次のサービス要求に影響を与えるかどうかである。ステートフル方式は、サービスされた要求の履歴に依存しており、ステートレス方式は依存していない。

ステートフルファイルサーバ

ステートフルファイルサーバは、クライアントの状態情報を1つのアクセス要求から次の要求まで維持する。つまり、ファイルをアクセスするためにクライアントから作られた2つの連続する要求に対して、クライアントに対して実行されたサービスに関連する状態情報が最初の要求の実行結果としてサーバプロセスによって保持される。この状態情報は、2番目の要求が実行されるときに継続して使われる。ファイルサーバがクライアントの状態情報をどれだけ長く維持するかを決めることができるように、クライアントによるすべてのファイルへのアクセス要求はセッションと呼ばれるオープンとクローズ操作の間に実行される。サーバは、オープン操作の実行によりクライアントが新しいセッションを開始したとき、クライアントに関する状態情報を作成する。そして、セッションの間はずっと状態情報を管理している。最後に、クライアントがクローズ操作を実行してセッションを閉じたとき状態情報は消去される。ステートフルサーバがどのように動作するかを示すために、以下の操作が可能なバイト列ファイルのファイルサーバを考える。

Open(filename, mode)：この操作は、指定された mode で、filename によって識別されるファイルをオープンするのに用いられる。この操作をサーバが実行するとき、オープンされたすべてのファイルの状態情報を管理するために使われる file-table に、このファイルに関する欄を設ける。ファイル状態情報は、通常ファイルの識別子、オープンモード、そして**読み込み書き込みポインタ (read-write pointer)** と呼ばれる非負整数ポインタの現在の値が含まれる。ファイルがオープンされると、その読み込み書き込みポインタの値はゼロが設定され、サーバはクライアントにファイル識別子 (fid) が返され、クライアントは続くファイルへの要求にこれを使う。

Read(fid, n, buffer)：この操作は fid で識別されるファイルから n バイトのデータを取り出して指定された buffer に入れるために使われる。サーバがこの操作を実行するとき、クライアントには読み込み書き込みポインタで指定されているバイトの地点から n バイトをクライアントに返し、そして読み込み書き込みポインタを n だけ増加させる。

Write(fid, n, buffer)：この操作の実行に当たり、サーバは n バイトのデータを指定された buffer から取り出して、それを fid で識別されるファイルに現在の読み込み書き込みポインタで指定される場所から書き込む。そして、読み込み書き込みポインタを n だけ増加させる。

Seek(fid, position)：この操作は、サーバに対して fid で識別されるファイルの読み込み書き込みポインタの値を指定された新しい値 position に変更させる。

Close(fid)：この命令は、サーバに対して fid で識別されるファイルの状態情報を file-table から削除させる。

上で述べたファイルサーバはステートフルである。なぜなら、クライアントによって利用されるためにオープンされたファイルの状態情報を保持しているからである。したがって、図 9.5 に示すように、ファイルをオープンした後で、クライアントが 2 つの連続する Read(fid, 100, buf) 要求を実行すると、最初の要求では最初 100 バイト (バイト 0 から 99) が読み込まれ、次の要求には次の 100 バイト (バイト 100 から 199) が読み込まれることになる。

ステートレスファイルサーバ

ステートレスファイルサーバは、クライアントの状態情報は一切管理しない。したがって、クライアントからの各要求には望んでいる操作をきちんと実行するために必要なすべてのパラメータを含んでいなければならない。つまり、各要求はファイルを指定して、読み込み／書き込みのためのファイル内の位置を指定する。例えば、バイト列のファイルに対して、以下の操作を可能とするサーバはステートレスサーバである。

Read(filename, position, n, buffer)：この操作の実行において、サーバはクライアントに対して filename で識別されるファイルの n バイトのデータを返す。返されるデータは buffer で指定される場所に置かれる。実際に記述されたバイト数もクライアント側に返る。読み込みを始めるファイル内の位置は、position パラメータで指定される。

Write(filename, position, n, buffer)：サーバがこの操作を実行するとき、buffer で指定されたところから n バイトを取り出して、それを filename で識別

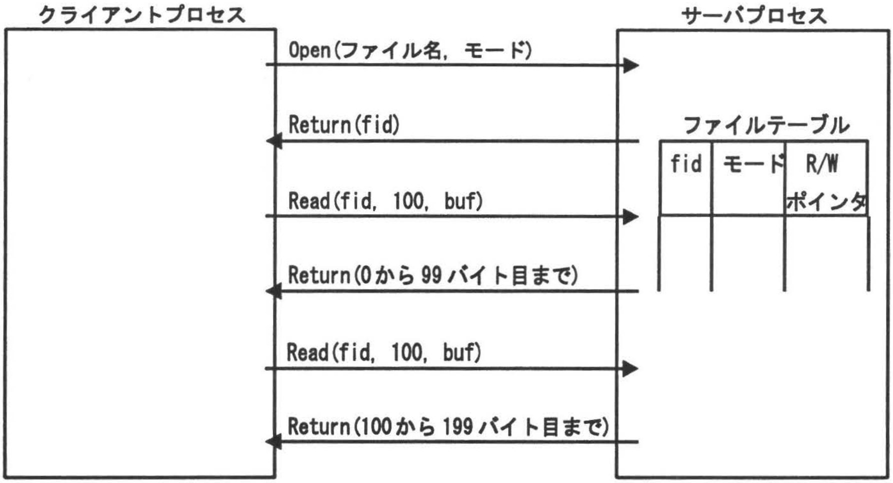


図 9.5 状態を持つファイルサーバの例

されるファイルに書き込む。position パラメータは、ファイルの中のどこから書き込むかを指定する。サーバは、実際に書き込まれたバイト数を返す。

図 9.6 で示されているように、このファイルサーバは以前に行われた操作によって生じたファイルのどんな情報も管理していない。したがって、もしクライアントが図 9.5 と同じ効果を望ならば、以下の 2 つの操作を実行しなければならない。

Read(filename, 0, 100, buffer)
Read(filename, 100, 100, buffer)

この場合、各ファイル操作は独立であるので、オープンやクローズ操作を使ったセッションの概念は用いる必要はない。しかし、図 9.6 に示したように、クライアントプロセスは通常使用しているファイルの状態情報を管理している。Sun Microsystems の NFS は、ステートレスサーバを使っている [Sandberg et al. 1985]。

クラッシュ回復でのステートレスサービスパラダイムの利点

多くの分散ファイルシステムによるステートレスファイルサーバの利用は、

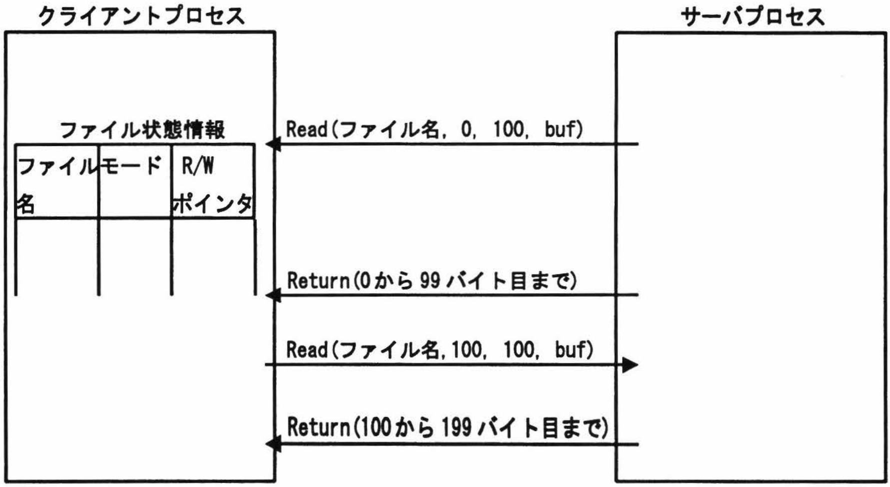


図 9.6　状態を持たないファイルサーバの例

障害が発生したときにステートレスサーバはステートフルサーバよりも大きな利点があるということが評価されている。例えば、ステートフルサーバでは、もしサーバがクラッシュして再起動したとき、そのサーバが持っていた状態情報が失われる。そして、クライアントプロセスはクラッシュに気づかずに処理を継続しようとし、一貫性のない状態を生じてしまう。同様に、クライアントプロセスがクラッシュして再起動したとき、サーバはもはや正しくないが簡単には取り出すことができない状態情報が保持されたままになる。したがって、ステートフルサービスパラダイムは複雑なクラッシュ回復手順を必要とする。クライアントとサーバの両方が、クラッシュを高い信頼性を持って検出できなければならない。サーバは、クライアントのクラッシュを検出することで、クライアントに関して保持しているすべての状態情報を消去できるし、クライアントはサーバのクラッシュを検出することで必要な誤り制御手順を実行することができる。

ステートレスサービスパラダイムは、クラッシュ回復を非常に容易にする。なぜなら、サーバによってクライアントの情報は一切管理されていないし、各要求には要求を実行するのに必要なすべての情報が含まれているからである。サーバが要求を処理している最中にクラッシュしたとき、クライアントはサーバが応答するまで再送するだけで良い。そして、サーバはクラッシュ回復は何

も行わない。クライアントが要求を処理している最中にクラッシュしたとき、クライアントでもサーバでも回復の手順は特に必要ない。したがって、ステートレスサーバは再実行可能な操作を構成することができる。つまり、もしクライアントが応答が返るまで要求を再送すると、サーバクラッシュによってデータが失われることはない。

しかし、ステートレスサービスパラダイムは、分散ファイルシステムの設計上以下の制約が課せられている [Silberschatz and Galvin 1994]。

1. ステートレスサービスパラダイムの各要求は、下位レベルのファイル識別子ではなく **filename** によってファイルを識別している。もし、遠隔の名前からローカルの名前への変換が各要求について行われると、要求処理のオーバヘッドが増加する。変換プロセスを避けるために、各ファイルはシステム上で唯一の下位レベルの名前を持つ必要がある。

2. クライアントによる要求の再送は、ステートレスサーバによってサポートされる操作が**繰り返し等価 (idempotent)** である必要がある。繰り返し等価な操作とは、何回それを繰り返して実行したかによらず、同じ結果となり同じ出力を返すことである。自己完結な読み込みと書き込み操作は繰り返し等価である。なぜなら、それらはファイル中の位置を示すのに絶対値のバイト数を使っており、増加するようなオフセットを使っていないからである。同様に、もしステートレスサービスパラダイムを使うときには、ファイルの削除のための操作も繰り返し等価になる必要がある。

ステートレスサービスパラダイムは、長い要求メッセージとなり要求の処理が遅くなるという欠点も持っている。要求メッセージは、望む操作を完全に実行するために必要なパラメータをすべて含む必要があるため、より長いものになる。他方、要求の処理は、ステートレスサーバが処理を高速化するための状態情報を一切持たないため、より遅くなる。更に、ある場合には、ステートフルサービスが必要となる。例えば、ネットワーク間ではメッセージが送られた順番では到着しないことがある。ステートフルサービスは、このような状況においては、管理された状態を使ってメッセージ順序を正しく並べ替えることができるため、有効である (第3章を参照)。同様に、ファイルシステムがサーバ主導のキャッシュ検証方式を使っている場合、ステートレスサービスパラダイムは使えない。なぜなら、サーバがどのファイルをどのクライアントがキャッ

シュしているかの記録を管理する必要があるからである (9.6.3節を参照)。

9.9　原子トランザクション

原子トランザクション (atomic transaction)(あるいは単に**トランザクション (transaction)**) は、障害の発生や並行計算においても分割されることなく実行される操作の集合からなる計算である。つまり、操作のすべてが正常に実行されるか、又はどの結果も残らないかのどちらかであり、並行に実行されている他のプロセスはこの計算の中間状態を変更したり見ることはできない。トランザクションは、障害や並行アクセスに対しても共有データオブジェクトの状態の一貫性を維持するのに役に立つ。トランザクションは2つの状態、つまりトランザクションが完全に実行されたか、完全に失敗したかという状態しかとらないため、回復が非常に容易になる。

トランザクションは、以下の基本的な特性を持っている。

1. **原子性 (atomicity)**:この性質は、外部の世界に対してトランザクションの操作が分割されずに実行されたように見えることを保証している。原子性のための2つの基本的な要求は、障害に対する原子性と並行アクセスに対する原子性である。**障害原子性 (failure atomicity)** は、もしトランザクションの作業が障害によって割り込まれた場合、すべての部分的に完了した結果は取り消されることを保証するものである。障害原子性は**すべてか無特性 (all-or-nothing property)** とも呼ばれている。なぜなら、トランザクションは通常完全に実行されるか全く実行されないかであるからである。他方、**並行原子性 (concurrency atomicity)** は、トランザクションが実行中に、トランザクションと並行に実行されている他のプロセスがトランザクションの中間状態を変更したり観測したりできないことを保証するものである。トランザクションが完了した後で最終結果のみが他のプロセスに見えるようになる。並行原子性は**一貫性特性 (consistency property)** とも呼ばれる。なぜなら、トランザクションは1つの一貫性のある状態から他の状態にシステムを移すものだからである。

2. **逐次化可能性 (serializability)**:この特性 (**独立性 (isolation property)** とも呼ばれる) は、並行実行されるトランザクションが互いに干渉しないことを保証している。つまり、2つ以上のトランザクションの集合の並行実行が、

それらを平行に実行したときの最終結果がある(システム依存の)順序で実行した場合と同じであるという意味で、**逐次等価 (serially equivalent)** である。

3. **不変性 (permanence)**:この特性 (**恒久性 (durability property)** とも呼ばれる) は、一度トランザクションが正常に完了すると、その結果は固定化され、対応するプロセスやそれが動作しているプロセッサがクラッシュしたとしても失われることがないことを保証するものである。

これらの性質を容易に思い出すために、Harder と Reuter[1983] は ACID という記憶方法を提案している。ここで、A、C、I そして D はそれぞれ原子性 (atomicity)(障害原子性)、一貫性 (consistency)(並行原子性)、独立性 (isolation)、そして恒久性 (durability) からきている。したがって、トランザクション特性は ACID 特性とも呼ばれる。

9.9.1 ファイルサービスにおけるトランザクションの必要性

ファイルサービスにおいてトランザクションの提供は2つの主な理由から必要とされている。

1. 障害に対してファイルの回復性を向上させるため。トランザクションの原子性によって、もしサーバやクライアントプロセスがトランザクションが終了する前にハードウェア故障やソフトウェア誤りで予測もなく停止してしまうとき、サーバは変更途中のファイルをすべて元の状態に復帰させることができる。トランザクション機能をサポートしていないファイルサービスにとって、操作の実行中のクライアントやサーバの予測できない障害は、変更途中のにあるファイルに一貫性のない状態を残すことになる。トランザクション機能を使わずに、現在の一貫性のない状態を元の状態へロールバック (回復) することは、ある場合には困難か不可能にさえなることがある。

2. 複数のクライアントによる変更可能なファイルの並行共有を一貫性のある手順で可能とするため。もし、複数のクライアントから同じファイルへのアクセスのためのアクセス要求が同期なしに実行されるとすると、異なるクライアントから要求された読み込みと書き込み操作の順序が何通りものインターリーブとなり、その中の幾つかはファイルを意図した状態とは違うものにすること

がある。したがって、複数のクライアントからのアクセス要求の非同期実行は、一般にファイルに予測できない結果を生じる。トランザクション機能は、基本的に共有されたファイルを意図した一貫性のある状態にするために、複数のクライアントからのアクセス要求を適切に逐次化する高レベルの同期機構である。

以下の例では、マシンや通信の障害、又は他のクライアントによるファイルへの並行アクセスなどの、クライアントの制御を越えたイベントによって生じるファイルの一貫性のなさを回避するために、ファイルサービスのトランザクション機能がどれだけ役に立つかを示している。

システム障害による不整合

図 9.7 の銀行トランザクションを考える。これは、口座 X から口座 Z に$5 を送金するための 4 つの操作 ($a_1$，$a_2$，$a_3$，$a_4$) からなる。顧客の口座記録はファイルサーバによってファイル内に管理されているとする。顧客口座記録への読み込みと書き込みのアクセスは、ファイルサーバにアクセス要求を送ることで行われる。トランザクション機能を除く基本ファイルサービスにおいて、口座 X から口座 Z に$5 を送金する処理は操作 a_1，a_2，a_3，そして a_4 をこの順番で実行することによってなされる。両方の口座の残高を$100 と仮定する。したがって、4 つのすべての操作が正しく実行されれば、口座 X と口座 Z の最終的な残高はそれぞれ$95 と$105 になる。ここで、操作 a_3 が正常に実行された後、操作 a_4 が実行される前にシステム障害が発生したとする。この場合、口座 X は引き出されているが口座 Z には預け入れが行われていない。したがって、口座 X と口座 Z の残高がそれぞれ$95 と$100 となり、$5 が消えることになる (図 9.8)。4 つの操作を再実行すれば、口座 X と口座 Z の最終的な残高は$90 と$105 となり、それは意図した結果とは異なる。

a1 : 口座 X の残高(x)を読み込む
a2 : 口座 Z の残高(z)を読み込む
a3 : (x－5)を口座 X に書き込む
a4 : (z＋5)を口座 Z に書き込む

図 9.7　口座 X から口座 Z に＄5 を送金するための操作の集合

他方、トランザクション機能を持つファイルサービスでは、4 つの操作 a_1,

図 9.7 における 2 つの口座の初期残高を $100 とする。

<table>
<tr><td>成功する実行</td><td colspan="2">成功しない実行</td></tr>
<tr><td>a1 : x=100
a2 : z=100
a3 : x=95
a4 : z=105</td><td colspan="2">a1 : x=100
a2 : z=100
a3 : x=95
システム障害</td></tr>
<tr><td>最終結果</td><td colspan="2">最終結果</td></tr>
<tr><td rowspan="2">x=95
z=105</td><td>4 つの操作が
トランザクション
として扱われない場合</td><td>4 つの操作が
トランザクション
として扱われた場合</td></tr>
<tr><td>x=95
z=100</td><td>x=100
z=100</td></tr>
</table>

図 9.8 図 9.7 の操作がトランザクション機能を使ったかどうによって、実行が成功した場合とそうでない場合の可能な最終結果

a_2, a_3, そして a_4 は 1 つのトランザクションとして扱われるため、それらは分割せずに実行される。この場合、もしトランザクションが正常に実行されれば、明らかに最終的な口座 X と口座 Z の残高は$95 と$105 となる。しかし、トランザクションが内部で失敗すれば、口座 X と口座 Z の最終の残高は$100 にロールバックされる。それは、障害が発生した時点での 2 つの口座の残高の中間状態がどうであったによらない (図 9.8)。したがって、障害が発生した場合、2 つの口座の残高は変更せずに保持され、トランザクションの再実行はどんな矛盾も生じない。

並行アクセスによる不整合

図 9.9 の 2 つの銀行トランザクション T_1 と T_2 を考える。トランザクション

a1：口座 X の残高(x)を読み込む
a2：口座 Z の残高(z)を読み込む
a3：(x－5)を口座 X に書き込む
a4：(z＋5)を口座 Z に書き込む

T1：口座 X から口座 Z に $5 を送金する

b1：口座 Y の残高(y)を読み込む
b2：口座 Z の残高(z)を読み込む
b3：(y－7)を口座 Y に書き込む
b4：(z＋7)を口座 Z に書き込む

T2：口座 Y から口座 Z に $7 を送金する

図 9.9　2 つの銀行トランザクション

T_1 は、口座 X から口座 Z に$5 を送金するもので、4 つの操作 a_1，a_2，a_3 そして a_4 からなる。同様に、トランザクション T_2 は、口座 Y から口座 Z に$7 を送金するもので、4 つの操作 b_1，b_2，b_3 そして b_4 からなる。2 つのトランザクションの実行による正味の結果は、次のようになる。

■口座 X の残高が$5 減る。
■口座 Y の残高が$7 減る。
■口座 Z の残高が$12 増える。

これらの 3 つの口座の初期残高を$100 と仮定すると、2 つのトランザクションを実行した後の口座 X、Y、そして Z の最終残高はそれぞれ、$95、$93、そして$112 となる必要がある。

トランザクション機能を持たない基本ファイルサービスにおいては、2 つのトランザクションに対応する操作が並行に実行されることが許されており、ファイルシステムがこれらの操作を逐次化することを行わないなら、予測していない最終結果が得られることがある。なぜなら、2 つのトランザクションに対応する操作の実行は、任意の順序でインターリーブされるからである。この

ような予測していない最終結果を生じるインターリーブの2つの例を図9.10に示す。この誤りの原因は、両方のクライアントが口座Zの残高をアクセスし、そしてその以前の値に依存する方法で変更を加えているからである。

トランザクション機能を持つファイルサービスでは、2つのトランザクションのそれぞれの操作は分割せずに実行され、どのトランザクションが先に実行されているかに依存せず正しい結果を生成する。したがって、トランザクション機能は並行アクセスによってファイルの一貫性を失うことなく複数のトランザクションの操作を逐次化する。しかし、同じデータをアクセスするすべてのトランザクションの完全な逐次化(1つのトランザクションは次のものが開始する前に完了する)は不必要な制約であり作業の完了に長い遅延を生じる。多くのアプリケーションでは、複数のトランザクションの一部には同時にインターリーブが可能でなお正しい結果を生成するものもある。例えば、図9.9における2つのトランザクションにおける操作の2つの可能なインターリーブで、正しい結果を生成するものを図9.11に示す。

2つ以上のトランザクションの操作の任意のインターリーブは、**スケジュール (schedule)** と呼ぶ。トランザクションがある逐次的な順序で1つ1つ実行したときと、同じ最終結果を生じるすべてのスケジュールを逐次等価という。逐次等価は、並行に実行されるトランザクションの正当性の基準として用いられる。並行に実行されるトランザクションの集合の実行を行うために逐次等価なスケジュールが常に選択されることを保証することがシステムに求められている。システムに操作の任意の順序を選択する自由を認めることにより、正しい最終結果を生成する順序が提供され、プログラマには自分で相互排他を行う必要をなくし、プログラミング作業を単純にすることになる。

9.9.2 トランザクションに基づくファイルサービスのための操作

トランザクション機能を提供するファイルシステムでは、トランザクションは読み込みや書き込みのような基本的なファイルアクセス操作の系列からなる。実際の操作とトランザクションを構成するそれらの系列はアプリケーションに依存しており、そのためトランザクションを構成するのはクライアントの責任となる。したがって、このようなファイルシステムのクライアントインタフェースはトランザクションサービスのための特別な操作が含まれている必要がある。トランザクションサービスのための3つの基本的な操作を以下に示す。

begin_transaction → returns(TID)：新しいトランザクションを開始し、個

図9.9の2つのトランザクションにあらわれた
3つの口座の初期残高を＄100とする

不正なスケジュール	別の不正なスケジュール
a_1:x=100	b_1:y=100
b_1:y=100	b_2:z=100
a_2:z=100	a_1:x=100
b_2:z=100	a_2:z=100
a_3:x=95	b_3:y=93
b_3:y=93	b_4:z=107
a_4:z=105	a_3:x=95
b_4:z=107	a_4:z=105
最終結果	**最終結果**
x=95	x=95
y=93	y=93
z=107	z=105

時間

図 9.10　図9.9の2つのトランザクションをインターリーブした2つの操作で、期待していない結果を生じたもの

別のトランザクション識別子 (TID) を返す。これは、このトランザクション内の他の操作で使われる。begin_transaction と end_transaction 内のすべての操作がトランザクションの本体を構成する。

end_transaction(TID) → returns(status)：この操作は、クライアントの視点からいうと、トランザクションが正常に完了したことを示す。したがって、トランザクションは終了され、コミットが行われる。返される状態は、トランザクションがコミットされたか、あるいはクライアント又はサーバによってアボートされて活動停止となったかを示す。もし、トランザクションがコミットされれば、すべての変更は固定化され、続くトランザクションがこのトランザ

図 9.9 の 2 つのトランザクションにあらわれた
3 つの口座の初期残高を $100 とする

正しいスケジュール	別の正しいスケジュール
a_1:x=100	b_1:y=100
b_1:y=100	b_2:z=100
a_2:z=100	a_1:x=100
a_3:x=95	b_3:y=93
a_4:z=105	b_4:y=107
b_2:y=105	a_2:z=107
b_3:z=93	a_3:x=95
b_4:z=112	a_4:z=105
最終結果	**最終結果**
x=95	x=95
y=93	z=93
z=112	z=112

時間 ↓

図 9.11 図 9.9 の 2 つのトランザクションをインターリーブした 2 つの操作で、期待した通りの結果を生じたもの

クションによって行われたファイルへのすべての変更結果を見ることができるようになる。他方、もしトランザクションがアボートされたなら、トランザクション内で要求された変更は 1 つも他のトランザクションには見えない。トランザクションは、クライアントからの明示的な要求と、トランザクションを破壊するシステム障害のイベントによってアボートされる。

abort_transaction(TID):トランザクションをアボートし、トランザクション内で実行された変更をもとの値に戻し、そしてその状態を活動停止とする。トランザクションは、通常あるシステム障害のイベントによりアボートする。しかし、クライアントがこのプリミティブを意図的にトランザクションをアボートするのに使うこともできる。例えば、図 9.12 のトランザクションを考える。これは、3 つのファイルへの書き込み操作からなっている。図 9.12(a) では、すべての 3 つの操作が正しく実行され、end_transaction 操作の後に、トランザク

ションは新しい値を他のトランザクションに見えるようにしている。しかし、図 9.12(b) では、3 番めの書き込み操作は十分なディスク空間がないために成功しなかった。したがって、この状況では、クライアントは abort_transaction 操作をトランザクションをアボートするために使うことができる。そのため、2 つの書き込み操作は取り消され、そしてファイルの内容はトランザクションが始まる前の状態に戻される。サーバで一度トランザクションがコミットされるかアボートされると、その状態はクライアントによってもサーバによっても戻すことはできない。結果的に、abort_transaction 要求をクライアントが出しても、トランザクションがコミットした後ではその要求が失敗する。

```
TID = begin_transaction
write(TID, file, position, n, buffer) ──▶ returns(ok)
write(TID, file, position, n, buffer) ──▶ returns(ok)
write(TID, file, position, n, buffer) ──▶ returns(ok)
end_transaction(TID)
```

(a)

```
TID = begin_transaction
write(TID, file, position, n, buffer) ──▶ returns(ok)
write(TID, file, position, n, buffer) ──▶ returns(ok)
write(TID, file, position, n, buffer) ──▶ returns(disk full error)
abort_transaction(TID)
```

(b)

図 9.12　*abort_transaction* 操作を使った処理、(a) すべての 3 つの操作は正常に実行されたのでトランザクションはコミットされた、(b) すべての 3 つの操作が正常に実行されなかったのでトランザクションはアボートされた。

トランザクション機能を持つファイルシステムでは、上で述べた 3 つのトランザクションサービス操作に加えて、トランザクションサービスにファイルアクセス操作がある。トランザクションサービスのファイルアクセス操作は、基本的なファイルサービス操作に対応する。トランザクションサービスのファイルアクセス操作のパラメータは、基本的なファイルサービスの操作と同じであるが、その操作が属しているトランザクションを識別するトランザクション識別子 (TID) が加えられている。例えば、9.8.2 節のバイト列ファイルに対するステートレスサーバにおける、トランザクションサービスのためのファイルアクセス操作を以下に示す。

Tread(TID, filename, position, n, buffer)：もし TID で指定されるトランザクションが何らかの仮の値を記録していれば、クライアントにはそのデータの n バイトが返る。そうでなければ、Read(filename, position, n, buffer) と同じである。

Twrite(TID, filename, position, n, buffer)：これは、Write(filename, position, n, buffer) と同じ効果があるが、TID がコミットしたときにのみ固定化される仮の形式で新しいデータを記録する。

9.9.3 回復技術

サーバの視点からは、トランザクションは 2 つのフェーズを持っている (図 9.13)。最初のフェーズはクライアントからの begin_transaction をサーバが受信したときに始まる。このフェーズでは、トランザクションのファイルアクセス操作が実行され、クライアントはファイルに変更を順次加えていく。end_transaction 又は abort_transaction 操作の実行によって最初のフェーズが終了して、2 番めのフェーズが始まる。2 番めのフェーズでは、トランザクションはコミットあるいはアボートのどちらかが行われる。コミットにおいては、トランザクションにおいてファイルに加えられた変更を固定化し、他のトランザクションから見えるようにする。他方、アボートにおいては、トランザクションによってファイルに加えられた変更を取り消し、トランザクションが開始する前の状態にファイルを戻す。

2 番目のフェーズにおいてトランザクションがコミットされるかアボートされるということから、最初のフェーズのファイル更新操作は永続的に記録されるか、取り消される場合があるような方法で実行される必要がある。したがって、トランザクションが最初のフェーズにあるとき、アボートに備えて、更新は取り消し可能な方法で記録されなければならない。取り消し可能な方法でファイルの更新を記録する一般的な 2 つの方法は、ファイル版方式と先書き込みログ方式である。

ファイル版方式

ファイルの回復性を保証する基本的な方法は、物理的な記憶にある実際のデータに上書きすることを避けることである。ファイル版方式はこの技術に基づいている。図 9.14 に示すように、この方法ではトランザクションが開始したとき、すべてのファイルアクセス操作 (トランザクション内の) がファイルを更

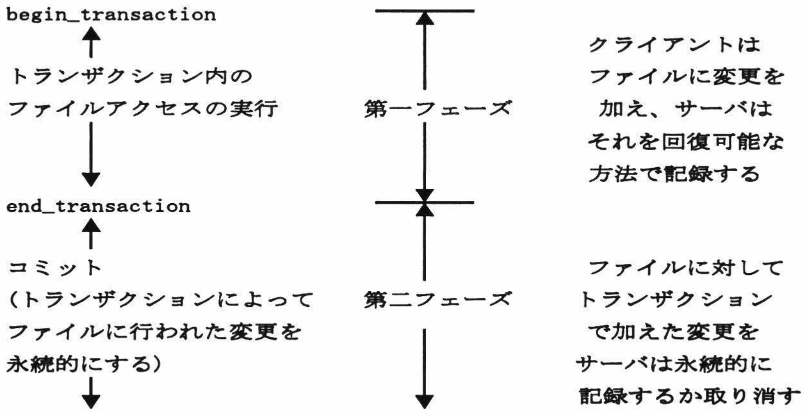


図 9.13　トランザクションの 2 つのフェーズ

新しないものは、現在のファイル版に実行される。トランザクションがコミットするとすぐに、ファイルに対して加えられた変更は公開される。したがって、ファイルの現在の版は最も最後にコミットされたトランザクションによって作られた版である。

トランザクション内でファイルを変更する最初の操作に到達したとき、サーバは現在のファイル版からそのトランザクションのために仮の版を作成し、このファイルの版に対する操作を実行する。この時点から、トランザクション内でこれに続くすべてのファイル操作 (読み込み、書き込み) はこの仮のファイル版に対して実行される。トランザクションがコミットされるとき、仮のファイル版が新しい現在の版になり、以前の現在版は古い版のリストに付け加えられる。他方、トランザクションがアボートされると、仮の版は簡単に捨てられ、現在の版が継続して現在の版となる。

1 つのトランザクションは 2 つ以上のファイルを更新することができる。この場合、トランザクションに対して各ファイルの仮の版ができることになる。更に、1 つのファイルが複数の並行トランザクションに含まれることもあり、同時に複数の仮の版が存在することもある。この場合、並行トランザクションの 1 つがコミットすると、そのトランザクションに関連する仮の版がファイルの現在の版になる。残った仮の版はもはや現在の版に基づいていないため、それらは次のように扱われる。並行トランザクションのうちの次のトランザクションがコミットするとき、もしこのトランザクションと以前コミットしたトラン

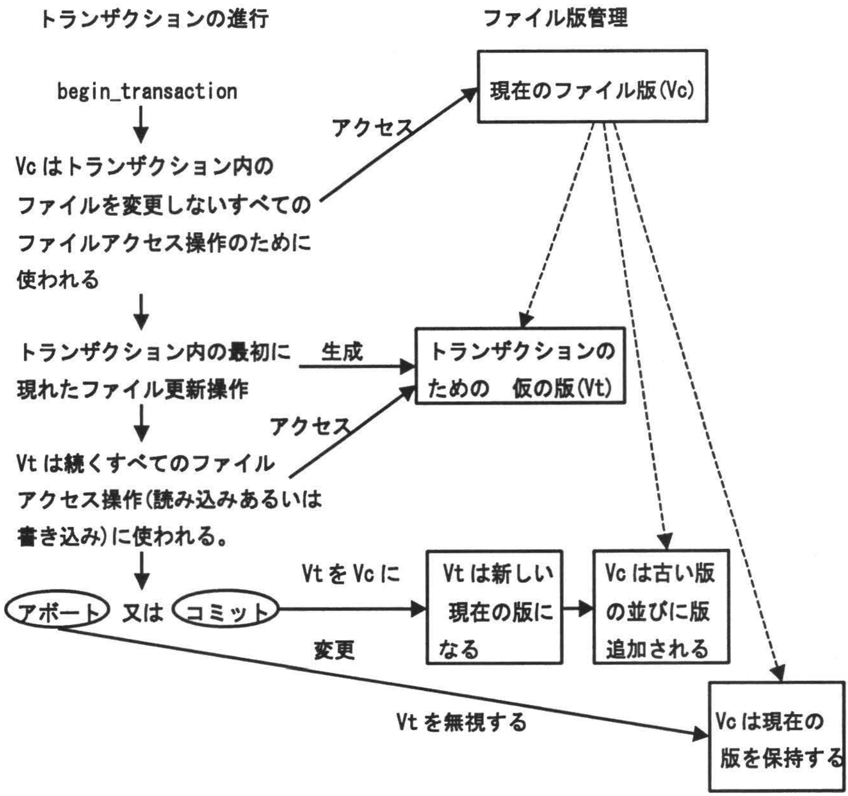


図 **9.14**　回復可能な方法でファイルの変更を記録するためのファイル版方式

ザクションの間に逐次可能性の矛盾が生じていないとき、このトランザクションに関連する仮の版は現在の版に統合され、既にコミットされたすべてのトランザクションの変更を含む新しい版が作成される。他方、もし逐次化可能性に矛盾があるとき、最初のトランザクションがコミットしたもの以外は、すべてのトランザクションがアボートされる。**逐次化可能性矛盾 (serializability conflict)** とは、2 つ以上の並行トランザクションがファイルにおける同じデータ項目をアクセスして 1 つ以上の操作が書き込み操作である場合である。

ファイル版の実装のためのシャドウブロック技術:ファイルの仮の版は、導出されたファイル版のコピーのように振る舞う。しかし、それを更新しようとするトランザクションに対してファイル全体をコピーすることは無駄であり、特に大きなファイルについては行うべきではない。シャドウブロック技術は、

ファイル全体をコピーすることなく、ファイルの仮の版を作成できる最適化手法である。実際、この方法はほとんどのコピーを削減している。

ファイルシステムでは、ファイルをディスク空間に確保するためにある索引機構を使っている。図 9.15(a) に示すように、この方法ではディスク空間はブロックと呼ばれる固定長のバイト列に分割されている。ファイルシステムは、各ファイルに対応する索引と空きブロックのリストを管理している。ある特定のファイルの索引は、ブロック番号とファイルデータが格納されるときに使われるその正確な順序を規定している。例えば、図 9.15(a) において、ファイル F_1 は番号 0，4，1 というブロックをこの順番で使っている。ファイル F_2 は番号 2，8，6，9，12 というブロックをこの順番で使っている。他方、空きブロックのリストには、現在空きになっていて新しいデータを格納するためにどのファイルに対しても確保することができるブロックの番号が管理されている。

シャドウブロック技術では、そのファイルの現在の版の索引をコピーすることによって、簡単に仮の版が作られる。つまり、ファイルの仮の索引が現在の索引から作られるということである。ここで、ファイル更新操作がブロックに影響が出ると、新しいディスクブロックが空きリストから取り出され、新しい仮の値がそれに書き込まれ、そして仮の索引における古いブロック番号は新しいブロック番号に置き換えられる [図 9.15(b) を参照せよ]。ファイルに新しい値を追加する更新操作についても、追加されるデータのための空きリストからの新しいブロックの確保と、ファイルにおける仮の索引の拡張によって同様に扱うことができる。ファイルの仮の版のために確保された新しいブロックを、**シャドウブロック (shadow block)** と呼ぶ。このファイルブロックに対して、これに続くトランザクションによる書き込みは、同じシャドウブロック上に実行される。

トランザクションにおけるすべてのファイルアクセス操作は、他のプロセスのファイルアクセス操作が現在の索引を用いて行われている間は仮の索引を使って実行される。したがって、トランザクションを実行しているプロセスは変更されたファイル版を見ており、すべての他のプロセスは元のファイル版を見続けている。変更されていないファイルブロックについては、これらの 2 つのファイル版で共有されていることを注意しておく。

図 9.15(b) に示されているように、もしトランザクションがアボートしたとき、ファイルの仮版のシャドウブロックは空きブロックリストに戻され、仮索引は単に消去される。他方、もしトランザクションがコミットするとき、仮の索引はそのファイルの現在の索引となり、そしてもし古いファイル版を保持する

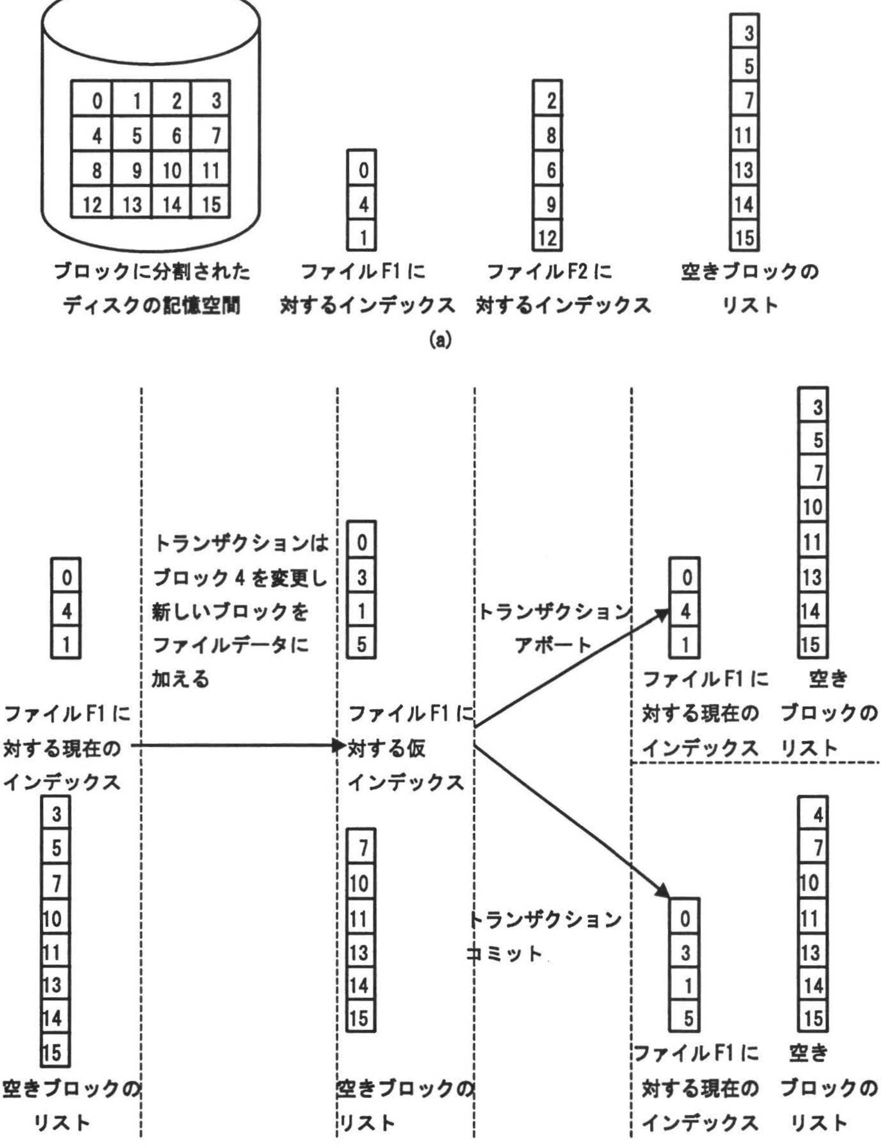


図 9.15 ファイル版を実現するシャドウブロック技法、(a) ディスクブロック、ファイルインデックス、そして空きブロックの例、(b) トランザクションの進展に伴うブロックの獲得と開放

必要がなければ、トランザクションによって変更されたデータを持つ元のファイルのブロックは空きブロックリストに加えられ、古い現在索引は消去される。

先書き込みログ方式

回復可能な方法でファイル更新を記録するもう 1 つのよく用いられる方法は、先書き込みログ方式である。この方法では、ファイルを更新するトランザクションの各操作に対して、記録が最初に作られて**先書き込みログ (write-ahead log)** と呼ばれるログファイルに書き込まれる。この後で、その操作は内容を変更するためにファイルに対して実行される。

先書き込みログは、安定記憶に保持されて、ファイルに対する変更を行う各操作についての記録が含まれる。各記録には、変更を行うトランザクションの識別子、変更が加えられるファイルの識別子、変更が行われる項目、各変更が行われる項目の古い値と新しい値が含まれる。

どのようにログが使われるかを説明するために、口座 X から口座 Z に$5 を送金するトランザクションについて考える。図 9.16 に示したように、ファイル項目に変更を加えるトランザクションの各操作に対して、記録が最初に作成されて、先書き込みログに書き込まれる。各記録の中には、操作によって変更される項目の古い値と新しい値がスラッシュで区切られて記述されている。

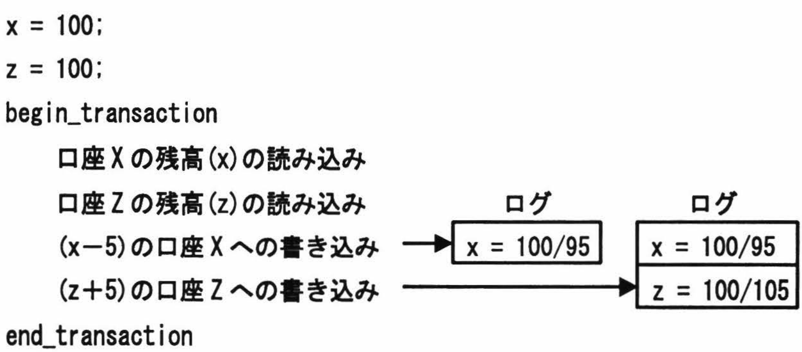


図 **9.16**　書き込み先行ログレコードの作成の例

トランザクションがコミットされるとき、コミット記録が先書き込みログに書き込まれる。更新操作による変更はファイル自身に行われるため、トランザクションがコミットするときはファイル項目は変更する必要はない。他方、トランザクションがアボートするとき、先書き込みログの情報が個々のファイル

項目をロールバックするのに使われ、そしてトランザクションによって変更されたファイルをその元の状態に戻す。ロールバックに対して、先書き込みログ記録は、そこに記述された変更を元に戻すために、最後の記録から後ろ向きに1つ1つ使われる。

先書き込みログは、クラッシュからの回復をも可能とする。例えば、図9.16のトランザクションを扱うサーバプロセスが、トランザクションでの2番めの書き込み操作でログ記録を書き込んだ後にクラッシュしたと仮定する。サーバのマシンが再起動した後で、先書き込みログが障害の正確な時点を識別するのに使われる。このため、ログの最後の記録における変更されたデータ項目(この場合z)の値が、ファイルの中の値と比べられる。ファイルの状態に応じて、以下の処理のうちの1つが行われる。

1. もしファイルの値が100なら、クラッシュはファイルが更新される前に起こっていることが分かる。そのため、ファイルの中のデータ項目の値は105に変更される。

2. 他方、ファイルの値が105であれば、ファイルが更新された後でクラッシュが起こったことを意味しており、この場合は何もする必要はない。

9.9.4 並行制御

逐次化可能性は、並行に実行されるトランザクションが互いに影響しないことを保証する、原子トランザクションの重要な性質である。しかし、もし同じデータ項目をアクセスしている2つ以上の並行トランザクションの操作が制御なしに処理を進めることを認めると、これらのトランザクションの操作は時間的に任意の順序でインターリーブされることになり、予測していない最終結果を生じることになる。したがって、複数のトランザクションの並行アクセスによるデータの一貫性のなさを回避するために、並行制御アルゴリズムの実現が必要となる。

良い並行制御機構は、共有データへの効果が逐次的に等価であるような手順でトランザクションが実行されることを保証し、最小のオーバヘッドで最大の並行性を引き出すものである。並行制御の最も簡単な方法は、トランザクションをある時間には1つしか動作しないようにするもので、2つのトランザクションが並行に動作することは決してないため、矛盾が生じることはない。しか

し、この方法では少しの並行性もないために良い方法ではない。1 つのトランザクションは、通常システムで利用可能なファイルの中のほんの少しのファイルしか使っていないために、並行性を削除することは認められないし必要でもない。この結果から、もう 1 つの簡単な並行制御の方法が得られる。2 つのトランザクションは、それらが共通のファイル (あるいは一般にデータ項目) を使っていないときには、並行に動作できるべきである。この方法は、前の方式よりは良くなっているが、1 組のトランザクションが同じデータ項目を少しの矛盾 (データ項目の一貫性のなさ) もない手順でアクセスすることもあるため、まだ不必要な制限がある。更に、1 つのトランザクションによってどのデータ項目が使われるかを予測するのは通常不可能である。最も一般に使われているものは、ロック、楽観的並行制御、そしてタイムスタンプである。これらの 3 つの方式を以下に述べる。

ロック

この方法は、最も古く最も広く使われている方法である。基本的なロック機構では、トランザクションはデータ項目をアクセスする前にそれをロックする。各ロックは、トランザクション識別子のラベルが付けられ、そのデータ項目をロックしたトランザクションのみがそれを何回でもアクセスすることができる。同じデータ項目をアクセスしようとする他のトランザクションは、データ項目がロック解除されるまで待たなければならない。トランザクションによってロックされたすべてのデータ項目は、トランザクションが完了した (コミットあるいはアボート) 時点で、すぐにロック解除しなければならない。ロックは、データアクセス操作の一部としてトランザクションサービスによって実行され、クライアントはデータ項目をロックしたりロック解除したりするための操作へはアクセスできない。

並行性の向上のための最適化ロック: 基本的なロック方式は制限的すぎるので、並行性を向上させるためにある最適化が提案されてきた。それらの中の 2 つを以下に示す。

1. タイプ別ロック (type-specific locking): データ項目へのすべてのアクセスのタイプに対して、1 つの簡単なロックしか使わないのは、必要以上の並行性を減らしている。並行性の向上は、アクセス操作のセマンティクスに応じて 2 つ以上のロックのタイプを使う、タイプ別のロック方式を使って達成できる。例えば、簡単なケースとして、読み込みと書き込みの 2 つのタイプのアク

セス操作について考える。同じデータ項目を読み込むが、決して書き込むことがない並行トランザクションでは、矛盾が発生しない。したがって、トランザクションがデータ項目を読み込み専用でアクセスしているときに、そのデータ項目を読み込み専用モードでアクセスしようとするトランザクションを待たせる理由はない。そのため、1つのロックを読み込みと書き込みの両アクセスに使うよりは、別々のロック (読み込みロックと書き込みロック) をそれらの2つの操作に対して使うべきである。これらの2つのタイプのロックを使うときの、ロックのルールは図 9.17 に示されている。データ項目に読み込みロックがかかっているとき、他の読み込みロックは許可されるが、書き込みロックは許可されない。したがって、読み込みロックがかけられている項目は、他のトランザクションによって変更することはできない。他方、書き込みロックがかけられているとき、どんな種類のロックも許可されない。したがって、書き込みロックがかけられている項目は、他のトランザクションによってアクセス (読み込み又は書き込み) されることはない。

既にセットされたロックの種類	セットされるべきロックの種類	
	読み込み	書き込み
なし	許可	許可
読み込みロック	許可	不許可
書き込みロック	不許可	不許可

図 9.17　読み込みロックと書き込みロックの場合のロック設定規則

2. **書き込み意志ロック (intention-to-write locks)**: トランザクションは2つのフェーズを持っており、データ項目に対するトランザクションによる更新は最初のフェーズで仮に行われ、2番目のフェーズでのみ固定化される。各トランザクションは、他のトランザクションの仮の値を見ることはできない。したがって、読み込みロックがセットされると、ロックされたデータ項目が他のトランザクションによって書き込まれることを防ぐことができるが、トランザク

既にセットされたロックの種類	セットされるべきロックの種類		
	読み込み	書き込み意志	コミット
なし	許可	許可	許可
読み込み	許可	許可	不許可
書き込み意志	許可	不許可	不許可
コミット	不許可	不許可	不許可

図 9.18　読み込みロック、書き込み意志ロック、そしてコミットロックの場合のロック設定規則

ションはそれがコミットできるまでの仮の書き込みを進めることは認められるべきである。その項目の値は、書き込みトランザクションがコミットされるまでは変更されないので、もしそれがその時点で停止していれば、その項目は読み込みトランザクションがそのロックを解放するまで変更されずに残される。

上で述べたことから、並行性の向上のために Gifford[1979b] は書き込み意志ロック (intention-to-write lock: I-write) とコミットロックを書き込みロックの代わりに提案した。したがって、この方式では 3 つのタイプのロックが使われ、ロックの規則は図 9.18 に示されている。もし、読み込みロックがセットされているとき、書き込み意志ロックは許可され、その逆も認められる。これは、書き込み結果は、書き込みトランザクションがコミットされないかぎり、他のトランザクションには見えないからである。もし、書き込み意志ロックがセットされていると、同じデータ項目に書き込み意志ロックをかけることはもはやできない。コミットロックは、もし他のタイプのロックが既にデータ項目にセットされていれば、許可されない。したがって、書き込み意志ロックを持つトランザクションがコミットするとき、その書き込み意志ロックをコミットロックに変更する。そのため、もし他の独立した読み込みロックがある場合、そのトランザクションはコミットロックをかけることができるまで待つことになる。

2 フェーズロックプロトコル:並行性を向上させるために、トランザクションが実際にそのデータ項目を使って作業する期間だけロックし、かつアクセス操作が完了した時点ですぐに解放するようにロックを使うことが望ましい。しかし、必要になったとき (必要でなくなったとき) にデータ項目をロックすること (ロックを解放すること) は、一貫性をなくす問題が生じる。例えば、読み込み

ロックと書き込みロックの早い解放 (アクセス操作が完了した時点での素早い解放) によって生じる 2 つの一般的な問題を以下に示す。

1. 同じトランザクションによる 2 つ以上の読み込みの場合に、読み込みに一貫性がなくなる問題。これは、読み込みロックと書き込みロックが早期に解放されると、トランザクションによる同じデータ項目に対する 2 つの連続する読み込みアクセス間で、もう 1 つのトランザクションが同じデータ項目を更新してコミットする可能性があるからである。したがって、2 回目のデータ項目への読み込みは、1 回目の読み込みと同じ値とはならないことがある。もしトランザクションがデータ項目を 1 回だけしか読み込まないならば、読み込みロックを早期に解放しても問題はない。

2. カスケード型アボートの必要性。トランザクションが書き込みロックを早期に解放して、他のトランザクションが同じデータ項目をロックし、その作業をすべて実行し、最初のトランザクションが終了する前にコミットしたとする。その後で、最初のトランザクションがアボートしたとする。この場合、既にコミットしたトランザクションは、その結果が決して見ることができないデータ項目の値に基づいていることから、取り消されなければならない。あるトランザクションがアボートしたことによって、既にコミットされているトランザクションがアボートすることを**カスケード型アボート (cascaded aborting)** という。カスケード型アボートには大きなオーバヘッドが生じるため、それを避けることが必要である。

データの一貫性がなくなる問題を避けるために、トランザクションシステムは 2 フェーズロックプロトコルを使う。トランザクションの最初のフェーズは、**拡大フェーズ (growing phase)** と呼ばれ、すべてのトランザクションに必要なロックが徐々に獲得される。次に、トランザクションの 2 番目のフェーズは**縮小フェーズ (shrinking phase)** と呼ばれており、獲得したロックが解放されていく。したがって、トランザクションはそのロックのどれかを一度解放すると、同じデータ項目あるいは別のデータ項目についてロックをかけることはできない。もし、すべてのトランザクションが 2 フェーズロックプロトコルを使っていれば、それらの操作のインターリーブによって形成されるすべてのスケジュールは逐次化可能であることが [Eswaran et al. 1976] によって証明されている。

ロックの粒度:ロックの粒度とは、ロック可能なデータ項目の単位を指している。トランザクションをサポートしているファイルシステムでは、この単位は普通はファイル、ページ、又はレコードである。もし多くのトランザクションがファイルを共有しているとき、ロックの粒度はどれくらいトランザクションが並行に実行できるかに大きな影響がある。例えば、ロックがファイル全体にしか適用できないとき、2 つの異なるトランザクションが同じファイル内にある関連のない 2 つのデータ項目をアクセスする場合に**偽共有 (false sharing)** が生じる可能性が高くなる (図 9.19)。このような場合、2 つのトランザクションによって 2 つのデータ項目が並行にアクセスできるにも拘わらず、ロックの粒度がファイルであるために認められない。ロックの粒度は偽共有の可能性を減らすことにより、並行性を向上させることができる。しかし、細かすぎるロックの粒度は大きなロック管理のオーバヘッドを生じ、回復可能なファイルの複雑な実装を生じ、そしてデッドロックの可能性を大きくする。

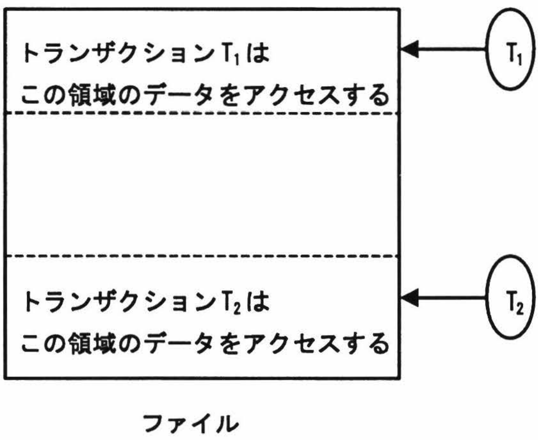


図 9.19　偽共有

デッドロックの扱い: ロック方式には、デッドロックを生じる可能性がある。**デッドロック (deadlock)** とは、トランザクションが他のトランザクションによってロックされているデータ項目を待っている状態である。ここで、このトランザクションは、他の待ちトランザクションを通したチェーンになっているかもしれないが、最初のトランザクションがそのロックを解放するのを待っている状態である。トランザクションは、それが終了するまでどのロックも解放

しないため、このような環状の待ちに入ったトランザクションはどれもその中のどれかがアボートしない限り先に進まなくなる。例えば、2つのトランザクション T_1 と T_2 がデータ項目 D_1 と D_2 をそれぞれにロックするとする。各トランザクションが他のトランザクションがアクセスする必要のあるデータ項目を持っているため、デッドロックが生じる。

デッドロックを扱うための一般に使われる技術の詳細は6章で述べられている。トランザクションにおいては、以下の技術の1つが用いられる。

1. **回避**:デッドロック問題を扱う1つの方法は、デッドロックを避けることである。デッドロックは、データ項目をロックする要求が予め定義された順序で行われるように強制することで、誰が誰を待っているかのグラフにサイクルができないので、回避できる。この方法は非常に効果的であるが、データ項目が早すぎるロックが行われる可能性があり、結果的に並行性が減少する。

2. **検出**:デッドロックは、誰が誰を待っているかのグラフを構成し検査することによって検出できる。このグラフにサイクルがあればそれはデッドロックの存在を示している。このようなサイクルが検出されたとき、サーバはサイクルに含まれる1つのトランザクションを選択してアボートさせなければならない。

3. **タイムアウト**:各ロックにタイムアウトを付けることはデッドロックを扱うもう1つの方法である。つまり、ロックはある固定された期間を超えることはできず、それを越えた後、ロックは効果がなくなる。効果がなくなったロックを持つデータ項目は、もしそのデータ項目のロックの解放を待っているトランザクションがない場合ロックされたままとなる。そうでなければ、ロックは壊れ (データ項目はロックが解放され)、待っているプロセスはそのデータ項目をアクセスするためにロックすることが認められる。ロックが壊されたトランザクションは、通常アボートされる。

タイムアウト方式の3つの大きな欠点は、(a) ロックのためのタイムアウト期間の長さを決定することが難しいこと、(b) 過負荷システムにおいて、タイムアウトによりアボートされるトランザクションが増加し、結果的にアボートされたトランザクションの再実行のオーバヘッドが増加する結果となること，(c) 長時間かかるトランザクションはよりペナルティを受けやすくなるために、この方法は長いトランザクションより短いトランザクションを優先するようになることである。

楽観的並行制御

Kung and Robinson[1981] の並行制御の方式は、トランザクションの並行実行によるアクセスの矛盾は非常にまれだという観測に基づいている。したがって、この方式ではトランザクションは最初のフェーズまでは制御無しに実行することを認めている。しかし、2 番目のフェーズでは、トランザクションがコミットする前に、そのデータ項目がそのトランザクションが開始する以前に変更されていないかどうかが検証される。もし正しいことが分かれば、トランザクションはコミットされる。そうでなければ、トランザクションはアボートされる。

検証過程において、トランザクションによってアクセスされているデータ項目の 2 つの記録が管理される。トランザクションによって読み込まれるデータ項目からなる読み込み集合と、トランザクションによって変更され、作成され、削除されるデータ項目からなる書き込み集合である。トランザクションを検証するために、その読み込み集合と書き込み集合とを、第一フェーズの終了に到達したすべての並行トランザクションの書き込み集合とを比較する。もし、検証されているトランザクションの読み込み集合又は書き込み集合に含まれるデータ項目が、上で触れた並行トランザクションの書き込み集合にも存在しているとき、検証は失敗する。

楽観的な並行制御の 2 つの主な利点は以下の通りである。

1. すべてのトランザクションはロックを待つ必要がなく独立に並行に処理を進められるために、最大の並行性を認められる。

2. デッドロックがない。

しかし、以下の欠点も存在する。

1. 最近コミットされたトランザクションに対応する古い版を検証プロセスのために保持しておくことが必要である。これは、ロック方式やタイムスタンプ方式では必要ではない。

2. この方式では、デッドロックは存在しないが、トランザクションの飢餓状態が生じることがある。なぜなら、検証に失敗したトランザクションはアボートされ、すべてがもう一度再開されるからである。もし、トランザクションが

再開されるたびにデータ項目の使用について他のトランザクションと矛盾が発生すると、それは決して検証に合格しないことになる。

飢餓状態問題を解決するために、Kung と Robinson はサーバが数回アボートされているトランザクションを検出することを提言している。このようなトランザクションが見つかったとき、セマフォによって保護された臨界領域を使うことによってデータ項目の排他的アクセスを与えるべきである。

3. 過負荷のシステムにおいては、アクセスの矛盾によってアボートされるトランザクションはかなりの数に上り、結果的にアボートされたトランザクションを再実行するためのオーバヘッドを増加させることになる。楽観的な並行制御は、このような状況には適していない。

Mullender と Tanenbaum[1985] は、幾つかのファイルが変更され、そして 2 つのトランザクションが同じデータ項目を使用するする機会が多いトランザクションについては、ロック方式を使うべきであると提言している。他方、楽観的並行制御は、1 つのファイルを用いて 2 つのトランザクションによる同じデータ項目へのアクセスが少ないと思われるようなトランザクションに使うべきである。

タイムスタンプ

楽観的な並行制御方式では、最初のフェーズが完了した後でのみトランザクションが検証される。もし、幾つかの矛盾が検出されて検証が失敗すれば、そのトランザクションはすべてアボートされてリスタートされる。矛盾を生じる操作が後続するトランザクションにおける操作の実行は、この場合無駄になる。矛盾を生じる操作を後続するトランザクションの操作の実行によるオーバヘッドは、もし操作が実行されるときに、矛盾があることが正しく検出できる場合無視できる。なぜなら、トランザクションは矛盾が発生したときにすぐにアボートされるため、そしてそれはすぐにリスタートされるためである。これは、タイムスタンプ方式によって可能とされてきた。この方法では、トランザクションにおける書く操作がそれが実行されるときに検証される。もし、検証が失敗すれば、トランザクションはすぐにアボートされリスタートすることができる。

操作レベルでの検証を可能とするために、各トランザクションは別々のタイムスタンプが begin_transaction を実行したときに付けられる。更に、各データー

タ項目は、**読み込みタイムスタンプ (read timestamp)** と**書き込みタイムスタンプ (write timestamp)** を持っている。トランザクションがデータ項目にアクセスするとき、アクセスのタイプ (読み込み又は書き込み) によってデータ項目の読み込みタイムスタンプ又は書き込みタイムスタンプが更新される。

通常、トランザクションの書き込み操作は、途中経過として記録され、他のトランザクションにはコミットするまで見えない。したがって、トランザクションが進行中のとき、いくつもの途中経過の値や書き込みタイムスタンプが存在することになる。途中経過の値やタイムスタンプは、トランザクションが終了した時点で固定される。

データ項目に対する読み込み操作又は書き込み操作を実行する前に、サーバはデータ項目のタイムスタンプを見て検証を行う。これには、完了していないトランザクションによる途中経過の値も含まれる。検証の規則を以下に説明する。

書き込み操作の検証 (validation of a write operation):もし、着目するトランザクション (書き込み操作を要求するトランザクション) が、アクセスされるデータ項目の読み込み及び (コミットされた) 書き込みのタイムスタンプと等しいかより新しい場合、書き込み操作は検証に合格する。したがって、この場合、途中経過の書き込み操作は実行される。他方、もし着目するトランザクションのタイムスタンプがデータ項目の最後の読み込み、あるいはコミットされた書き込みよりも古い場合、その検証は失敗する。なぜなら、着目するトランザクションが開始した後で、他のトランザクションがアクセスしているからである。したがって、この場合着目するトランザクションはアボートされる。

読み込み操作の検証 (validation of a read operation):もし着目するトランザクション (読み込み操作を要求したトランザクション) のタイムスタンプが、アクセスされるデータ項目のすべてのコミットされたあるいは途中経過の値の書き込みタイムスタンプよりも新しいとき、データ項目の途中経過の値がないときはすぐに読み込み操作が実行される。そうでなければ、データ項目についての途中経過の値を持つトランザクションが完了するまで待たなければならない。他方、以下の場合は検証は失敗し、現在のトランザクションはアボートされる。

1. 着目するトランザクションのタイムスタンプが、データ項目に対する最も新しい (コミットされた) 書き込みよりも古い場合。

2. 着目するトランザクションが、固定のデータ項目のタイムスタンプよりは新しいが、もう1つのトランザクションによって作られた途中経過のデータ項目より古い場合。

上で述べられた方法では、もしすべての操作が以前のトランザクションと一貫性を持つためには、最初のフェーズしか実行できない。したがって、そのトランザクションが最初のフェーズを実行した後、以前のトランザクションが共有するデータ項目の途中経過の値をコミットするまで待つ必要があるが、通常コミットできる。

上で述べたタイムスタンプに基づく並行制御は、$SDD-1$データベースシステム [Bernstein et al. 1980] に使われている。タイムアウトを使っている同様な方法は、[Reed 1983] に論じられている。タイムスタンプに基づく並行制御方式はデッドロックを発生しない。

9.9.5 分散トランザクションサービス

トランザクション機能を持つ分散ファイルシステムは、分散トランザクションサービスをサポートする必要がある。分散トランザクションサービスは、従来のトランザクションサービスの拡張である。それは、トランザクションに含まれるファイルが2つ以上のサーバによって管理されているものである。トランザクションが複数のサーバを含むとき、すべてのサーバは、トランザクションに含まれるすべてのファイル操作が回復可能性と並行制御を達成するために、トランザクションの処理の間その動作を他のサーバと調整するために互いに通信する必要がある。

分散トランザクションに含まれている複数のサーバの動作を調整する簡単な方法は、すべてのクライアントからの要求を強制的に1つのサーバを通して発行することである。しかし、不必要な通信オーバヘッドを避けるために、分散トランザクションサービスは通常1つのサーバを通して要求する代わりに直接にファイルを持つサーバにクライアントからの要求が送られる。この方法は、以下に述べる。この考え方に基づく記述は [Israel et al. 1978] にあり、そして XDFS ファイルサービス [Mitchell and Dion 1982, Sturgis et al. 1980] に使われている。

分散トランザクションサーバにおいては、クライアントが begin_transaction 要求をすべてのサーバに送ることによってトランザクションが始まる。連絡さ

れたサーバは、begin_transaction 要求を実行しその結果の TID をクライアントに返す。このサーバはトランザクションの**調整者 (coordinator)** となり、アボートあるいはコミットの責任を持ち、作業者と呼ばれる他のサーバを追加することを行う。

作業者は、動的にトランザクションに加えられる。このため、分散トランザクションは従来のトランザクションに加えて新しい操作を持っている。

add_transaction(TID, server_id of coordinator)

この要求は、サーバが TID の内部に含まれることを通知している。

トランザクションに加入していないサーバにアクセス要求を送信する前に、add_transaction 要求がサーバに送られる。サーバが add_transaction 要求を受けると、そのサーバは調整者のサーバ識別子を記録し、TID を含む新しいトランザクションの記録を作成し、そしてそのトランザクションからのローカルファイルの更新を記録する新しいログを初期化する。そして、調整者に対してトランザクションに加入する意図を通知する連絡を行う。この手順において、各作業者は調整者のことを認め、調整者はそのトランザクションに含まれるすべての作業者のリストについて認めることになる。この情報は、トランザクションの調整者と作業者に対してコミット時に互いに調整することを可能としている。

2フェーズマルチサーバコミットプロトコル

分散トランザクションの最も重要な部分は、分散トランザクションのコミットである。分散トランザクションにおいては、トランザクションによって変更されるファイルが複数のサーバに格納されているため、トランザクションプロトコルはより複雑になる。1つのサーバのクラッシュが通常他のサーバに影響しないことから、コミットプロトコルは、トランザクションがもしすべての含まれているサーバが完了できない場合はコミットせず、幾つかのサーバに対する更新も変更されることを保証する必要がある。

分散トランザクションをコミットするための一般的なプロトコルは2つのフェーズを持っている。2フェーズマルチサーバコミットプロトコルは [Gray 1978] によって以下のように提案された。

分散トランザクションのクライアントが end_transaction 要求を作成するとき、トランザクションにおける調整者と作業者は自身のファイルに影響する操作を記述したログの途中の値を持っている。調整者は、トランザクションがア

ボートされるべきかコミットされるべきかの決定を行う責任を持っている。もし、どれかのサーバがコミットできないときは全体のトランザクションはアボートされなければならない。そのため、end_transaction 操作は 2 つのフェーズで実行される。それは、準備フェーズとコミットフェーズである。各フェーズに含まれる動作は次の通りである。

準備フェーズ (preparation phase)

1. 調整者はその中にコミットプロトコルが開始したことを記録するログを作成する。

2. 次に、すべての作業者に prepare メッセージを送り、コミットする準備を作業者に行わせる。そのメッセージにはタイムアウト値もついている。

3. 作業者がそのメッセージを受信すると、もしそれがコミットする準備ができているかどうかを検査する (例えば、そのトランザクションの一部が前にアボートされていないなど)。もしそうであれば、ログにそのことを記録し、ready メッセージを応答する。そうでなければ、abort メッセージを応答する。

コミットフェーズ (commit phase):この時点で、調整者は各作業者から ready か abort の応答を受け取るか、prepare メッセージがタイムアウトとなる。

1. もしすべての作業者がコミットの準備が整えば、トランザクションはコミットされる。これに対して、調整者はログにトランザクションがコミットされたということを示す欄を作成する。そして、作業者にコミットするようにコミットメッセージを送る。この時点で、トランザクションは事実上完了し、調整者は結果をクライアントに報告する。

他方で、どれかの応答が abort であるか、どれかの作業者への prepare メッセージがタイムアウトしたとき、トランザクションはアボートされる。これに対して、調整者はトランザクションがアボートされたことを示すログを含む欄を作成する。そして、作業者にアボートを依頼するメッセージを送り、クライアントに失敗を報告する。

2. 作業者が commit メッセージを受信すると、そのログに committed という欄を作成し、調整者に committed を応答する。この時点で、作業者における

トランザクションの部分は完了し、作業者によって保持されていた記録は消去される。

3. 調整者がcommittedをすべての作業者から受信すると、トランザクションは完了したと考えられ、調整者によって保持された記録はすべて消去される。調整者はすべての作業者からcommittedの応答が返るまで、commitメッセージを再送し続ける。

9.9.6　入れ子トランザクション

入れ子トランザクション (nested transaction) は、従来のトランザクションの一般化であり、他のトランザクションの一部となっているトランザクションを**下位トランザクション (subtransaction)** という。下位トランザクションは、それ自身が下位トランザクションを持っても良い。この場合、トランザクションは任意の深さに入れ子にすることができ、トランザクションのファミリを形成する。

同じファミリに属するトランザクション間の関係を記述するのに、木表現が通常用いられる。トランザクションが開始するとき、それは**トップレベルトランザクション (top-level transaction)** と呼ばれる1つのトランザクション(プロセス)からなる。このトランザクションは、子供を生成することができ、下位トランザクションを生じる。これらの子供は、再度それ自身の子供を生成してその下の下位トランザクションを生じることができる。あるトランザクションが下位トランザクションを生成したとき、それを下位トランザクションの**親 (parent)** と呼び、その下位トランザクションを**子供 (child)** と呼ぶ。**祖先 (ancestor)** と**子孫 (descendants)** も使われる。あるトランザクションは、それ自身の祖先であり子孫である。

入れ子トランザクションのコミット

入れ子トランザクションシステムにおいては、その子孫がすべてコミットしたときに限ってトランザクションはコミットできる。しかし、トランザクションはいつでもアボートできる。そのため、全トランザクションファミリがコミットするためには、ファミリ内の他のトランザクションがコミットすることをトップレベルのトランザクションが待つ必要がある。

下位トランザクションは、その親に対しては原子的に見える。つまり、それが実行する操作は、従来のトランザクションと同じように障害や並行計算の両

者に関して分割できないように実行される。したがって、下位トランザクションによってデータ項目に行われる変更は下位トランザクションがコミットした後でのみ見えるようになり、そのことを親に通知する。その結果、下位トランザクションによって実行される更新の現実のコミットは、トップレベルトランザクションまではるばると続く各祖先トランザクションのコミットに依存することになる。

他方、もし障害が発生し下位トランザクションがその完了前にアボートしたとき、その途中の更新のすべては取り消される。そして、それが親に通知される。親は、処理を継続して代わりの方法を使って作業を完了させることを試みるか、自身もアボートするかを選択することになる。したがって、下位トランザクションのアボートは必ずしもその祖先のアボートにはならない。しかし、もし障害によって祖先のトランザクションがアボートした場合、その子孫のすべてのトランザクションの更新 (既にコミットされているもの) は取り消されなければならない。したがって、トップレベルトランザクションがコミットするまでは、全トランザクションファミリによって行われた更新は永続化はなされない。トップレベルトランザクションのコミットが成功した後でのみ、それが子供に通知される。

入れ子トランザクションの利点

入れ子トランザクションの機能は、以下の主な利点によって従来のトランザクション機能 (特に分散システムにおける) の重要な拡張と考えることができる。

1. トランザクションの内部に並行処理を可能とした。つまり、トランザクションは、異なるプロセッサで並行に動作する幾つかの下位トランザクションを生成することを可能としている。1つの親トランザクションのすべての子トランザクションは同期するため、親トランザクションはこの場合でも逐次化可能性を維持している。

2. 障害に対するより強い保護を与える。つまり、トランザクションの内部に検査ポイントを設けることを可能としている。なぜなら、親トランザクションの下位トランザクションは、親トランザクションや他のトランザクションに独立に失敗するからである。したがって、下位トランザクションがアボートしたとき、その親は継続してその作業を完了するために失敗した下位トランザクションの代わりに別の下位トランザクションを生成することもできる。

9.10　設計原理

Satyanarayanan[1992] による AFS と他の分散ファイルシステムの実験に基づいて、彼は分散ファイルシステムの設計のための一般的な原理を提唱した。

1. クライアントは燃焼サイクルを持つ。この原理は、もし可能であれば、サーバマシンで実行するよりもクライアントマシンで実行する方が望ましいということを述べている。なぜなら、すべてのクライアントの共通の資源であり、したがって、サーバマシンのサイクルはクライアントマシンのサイクルよりもより高価である。集中した (共通に使われる) 資源の必要性が増加することを教訓として、システムのサイズが大きくなるにつれて緩やかなシステム性能の低下を可能とするため、この原理は設計の拡張性の向上を目的としている。

2. いつでも可能なキャッシュ。性能の向上、拡張性、利用者の移動性、そしてサイトの自律性がこの原理の動機である。クライアントサイトでのデータのキャッシュは、それが現在使われているときはいつでもデータが利用可能であるため、計算時間とネットワークの帯域を大きく節約でき、全体のシステム性能を向上させることが多い。キャッシュは、集中する資源の競合を削減するので、拡張性も向上させる。

3. 利用特性を調べる。この原理は、利用の特性 (アクセスや変更パターン) に応じて、ファイルを容易に識別される小さな数のクラスにグループ化して、個々に最適化を行って性能を向上させるための、クラスに応じた特性を調べる必要があるということである。例えば、ファイルが高い頻度で読み込みが行われるが、書き込みは一度しか行われない場合、読み込み専用複製のための不変ファイルとして扱うことができる。システムプログラムのオブジェクトコードが入っているファイルなどは、このクラスの良い候補である。

性能向上のために、異なるクラスに属するファイルに対して異なる機構で扱うことは、ファイルシステムを複雑にする。そのため、設計の簡潔さから、設計者の一部は、すべてのファイルを扱うために 1 つの機構を使うことを望んでいる。

4. システム全体に渡る知識と変更の最小化。この原理は、設計の拡張性を向上させることを目的とする。分散システムが大きくなるに従って、常にシステ

ムのすべての状態に注意していることや、一貫した方法で分散した又は複製されたデータを更新することがより困難になってくる。したがって、グローバルな情報の監視や自動的な更新は、実用的には避けるべきである。キャッシュ検証のためのコールバック方式と、**アクセス制御リスト (ACL)** における否定権利の利用に基づくアクセス制御 (11 章で述べられる) はこの原理の 2 つの応用である。階層的なシステム構造も、この原理の応用である。

5. なるべく少ない実体のみを正当なものとする。この原理は、システムの安全性を向上させる目的を持っている。例えば、数千のクライアントよりは、非常に少ない数のサーバの一貫性に基づく安全性を保証するほうが簡単である。この場合、これらのサーバの物理的な安全性とそれらで動作しているソフトウェアの安全性を保証するだけで十分である。

6. もし可能ならバッチとする。バッチは、性能の向上にしばしば寄与する。例えば、操作のグループ化は、遅延は生じるが、スループットを向上させることができる。同様に、個々のページよりも大きな塊でネットワーク上にデータを転送したほうが、より効率的である。ファイル全体の転送は、この原理の応用である。

9.11 事例紹介：DCE 分散ファイルサービス

分散コンピューティングシステムにおける 2 つの一般的な商用ファイルシステムは、SUN Microsystems 社の**ネットワークファイルシステム (Network File System：NFS)** と、Open Software Foundation の**分散ファイルシステム (Distributed File System：DFS)** である。DFS は、**分散コンピューティング環境 (Distributed Computing Environment： DCE)** によって提供される多くのサービスのうちの 1 つである。

分散システムを構築するために、この章で述べた概念と機構がどのように使われているかのケーススタディとして、DFS について以下に説明する。NFS に関する良い解説は、[Khanna 1994, Sandberg 1987, Sandberg et al. 1985] にある。しかし、以下の DFS での説明には、NFS との違いについても指摘している。

DCE の DFS は、幾つかの魅力的な機能を持ち、将来の分散コンピューティング環境の主要な役割を果たすことを約束している。これは、CMU のアンド

リューファイルシステム (Andrew File System：AFS) を基にして作られているが、多くの新しい機能が備えられている。例えば、AFS は共有ファイルにセッションセマンティクスをサポートしているが、DFS はより精密な単一サイト UNIX セマンティクスをサポートしている。

DFS は、基本的には DCE の他のサービスを利用している DCE アプリケーションである。DFS は、複数のファイルアクセス要求を同時に扱うために DCE スレッドを用いている。また、ファイルアクセス操作のクライアント-サーバ通信のために**遠隔手続き呼び出し (Remote Procedure Call：RPC)** を用いている。DFS は、複数のサーバ間においてクロックの同期をとるために**分散タイムサービス (Distributed Time Service：DTS)** を用いており、そしてファイルアクセス要求のときにクライアント-サーバ間の認証や認定のためにセキュリティサービスを用いている。そして、簡単にファイル名だけですべてのセルにおけるすべてのクライアントによってシステム全体に存在するすべてのファイルがアクセスできるようにするための、すべてのファイルに対する単一でグローバルな名前空間を提供するためのディレクトリサービスを使っている。

DFS は、DCE システムの 1 つのノード上に同時に複数のファイルシステムの存在を認めるように設計されてきた。例えば、UNIX, NFS そして DFS 自身のファイルシステムがそのノードの利用者に対して 3 つの異なるファイルシステムを提供するために共存が可能である。単一のノード上に DFS を提供する DFS のローカルファイルシステムは Episode[Chutani et al. 1992] と呼ばれる。予期されるように、1 つのノードに複数のファイルシステムを共存させるときは、DFS に特有の機能は Episode ファイルシステムを使う利用者にのみ利用可能である。DFS の機能は、他のファイルシステムの利用者には利用できない。

9.11.1　DFS ファイルモデル

UNIX のように、DFS はファイルが構造化されていないデータのシーケンスとなっている非構造化ファイルモデルを用いている。DFS ファイルサーバは、ファイルの内容を解釈しないバイトの並びとして扱う。1 つのファイルは、2_42 バイトまでを格納できる。

UNIX のように、DFS は変更可能なファイルモデルも使うことができる。これは、各ファイルについて、更新操作によって改変される唯一の格納された並びが存在するものである。DFS は、変更可能なファイルモデルを使うが、**クローン化 (cloning)** と呼ばれる機能 (後で述べる) によって、1 つのファイルに

対して同時に2つの格納された並びを持つことを可能とする。この1つは、クローン化の前の版であり、もう1つはクローン化の後に加えられた変更を含むものである。

9.11.2　DFS ファイルシステムモデル

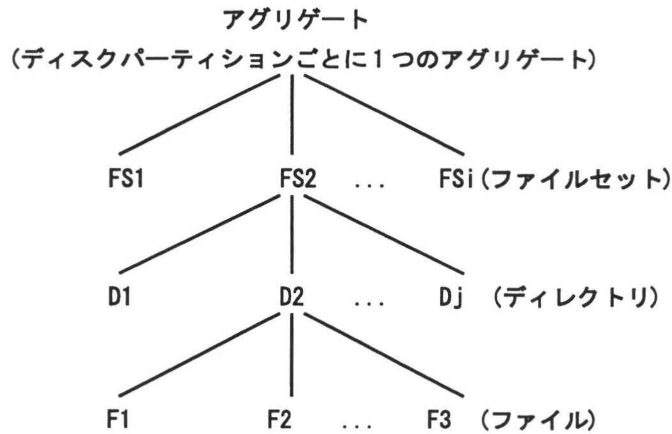


図 9.20　DFS ファイルシステムモデルにおける4つのレベルの集合体

図9.20に示すように、DFS ファイルモデルは4つのレベルの集合体を持っている。最も下位のレベルはファイルである。次のレベルはディレクトリである。各ディレクトリは通常幾つかのファイルを含んでいる。ディレクトリの上はファイルセットである。各ファイルセットは、通常幾つかのディレクトリを含んでいる。最後に、最も高いレベルはアグリゲートであり、通常幾つかのファイルセットを含んでいる。各ディスクパーティションには厳密に1つのアグリゲートが保持される。

UNIX のファイルシステムと同様に、ファイルセットは集合として管理される (移動、複製、バックアップなど) ファイルのグループである。しかし、UNIX とは異なりファイルセットは通常ファイルシステムのサブツリーであり、ファイルシステムツリーの全体ではない。例えば、ファイルセットは単一の利用者のファイルのすべてを含んでもよいし、関連する利用者のグループのファイルすべてを含んでもよい。この違いによって、DFS はディスクパーティションごとに複数のファイルセットを持つことができ、UNIX や NFS がディスク

パーティションごとに1つのファイルシステムしか持てないのに比べて管理上の利点がある。その主な利点は、ほとんど満杯のパーティションから相対的に空きのあるパーティションに必要な時に必要なだけファイルセットを移動することによって、異なるパーティションを動的にバランスをとることができるため、ディスクスペースを有効に利用できることである。

9.11.3　DFS ファイルアクセスモデル

分散ファイルサービスは、クライアント-サーバアーキテクチャに従っており、ファイルアクセスにはデータキャッシュモデルを使っている。DCE システムにおける1つのマシンは DFS クライアントであるか、DFS サーバであるか、その両者である。DFS クライアントとは、DFS サーバによって管理されるファイルセット内のファイルを使うマシンのことである。DFS クライアントマシンの主なソフトウェア部品は、**DFS キャッシュ管理者 (DFS cache manager)** であり、性能向上のため最近アクセスしたファイルの一部をキャッシュするものである。

もう一方では、DFS サーバは自分のローカルディスク上のファイルセット内にファイルを管理するための自身のディスク記憶を持っているマシンであり、DFS クライアントからのサービス要求を受け付ける。DFS サーバは以下のソフトウェア部品を持っている。

1. **Episode**: これは、DFS のローカルファイルシステムである。

2. **トークン管理者 (token manager)**:トークン管理者は、複数のキャッシュの一貫性を管理する問題を扱うためにトークンに基づいた方法を実現するために使われる (この方法は後で述べる)。

3. **ファイルエクスポータ (File exporter)**:ファイルエクスポータは、クライアントからファイルアクセス要求を受け付けて、それらの応答を返す。クライアントとファイルエクスポータとのやりとりは、DCE の RPC を使って行われる。Episode ファイルへの要求の扱いに加えて、ファイルエクスポータはそのノードに存在している他のすべてのファイルシステムへの要求を扱う。また、クライアントと DFS サーバとの安全な通信チャネルを確立するために、クライアントの認証についても扱う。ファイルエクスポータはマルチスレッド化さ

れているので、幾つかのクライアントからの要求を同時に扱うことができる。

4. **ファイルサーバ (file server)**:ファイルサーバは、ローカルのファイルセットを管理する。これは、ファイルセットが幾つ存在しているのか、そしてどのファイルセットがどのディスクパーティションに属しているかを管理する。また、ファイルセットの情報を得るため、またファイルセットのディスク割り当てを操作するため、そしてファイルセット全体を作成、削除、複製、移動、バックアップ、クローン化、そして再格納するためのシステム管理者用のコマンドを提供している。

5. **ファイルセット位置特定サーバ (fileset location server)**:ファイルセット位置特定サーバは、どの DFS サーバがセルにおけるどのファイルセットを管理しているかの情報を管理している。もし、ファイルセットがある DFS サーバから別のサーバに移ったとき、又は別の DFS サーバに複製されたとき、ファイルセット位置特定サーバはこれらの変化を記録する。ファイルの名前が与えられると、ファイルセット位置特定サーバはそのファイルが存在するファイルセットを管理している DFS サーバのアドレスを返す。もし、ファイルセットが複製されている場合、それを管理しているすべての DFS サーバのアドレスが返る。DFS クライアントがその名前を指定してファイルをアクセスするとき、そのキャッシュ管理者は、ファイルセット位置特定サーバからそのファイルのファイルセットを管理している DFS サーバのアドレスを得る。キャッシュ管理者はこの情報を将来の利用のためにキャッシュする。

6. **複製サーバ (replication server)**:複製サーバは、ファイルセットの複製の一貫性を管理する。

上で示した 6 個の DFS サーバの部品のうち、先の 3 個はカーネル空間に存在し、後の 3 個は利用者空間に存在する。

DFS クライアントがファイルアクセスを行うとき、クライアントのマシンのキャッシュ管理者は最初に要求されたデータが既にキャッシュに存在しているかどうかを検査する。もし、データがキャッシュにあれば、ファイルアクセス操作は DFS サーバに接続することなくローカルに実行される。そうでなければ、キャッシュ管理者は適切な DFS サーバにデータを要求する RPC を発行する。DFS サーバから受信したデータは、キャッシュ管理者によって将来の利用

のためにキャッシュされる。

DFSは、データ転送の単位として、ブロックレベル転送モデルを使っている。ブロックサイズは64キロバイトであり、かなり多くの(ある環境ではほとんどの)ファイルは全体が転送されキャッシュされる。

NFSにおけるファイルアクセスについてふれると、NFSもファイルアクセスにデータキャッシュモデルを使っており、データ転送の単位にブロックレベル転送を使っている。NFSのブロックサイズは8キロバイトである。性能改善のために、データキャッシュの他にNFSは**先読み (read-ahead)** 機構を用いている。この機構によると、クライアントマシンにあるファイルシステム部品が必要とするデータを含むブロックを受信した後に、自動的に次のブロックについての要求を発行し、それが必要とされたときには素早くローカルに利用可能とする。

9.11.4 DFSファイル共有セマンティクス

DFSの最も強力な機能は次の点である。データキャッシュモデルを使っているにも拘わらず、DFSは単一サイト UNIX セマンティクスをサポートしている。これは、1つのファイルに対する各読み込み操作が、以前に実行されたそのファイルに対する書き込み操作のすべての結果が見えるということである。これは、以下に記述する方法で達成されている。

各DFSサーバはトークン管理者と呼ばれる部品があることを既に述べた。トークン管理者の仕事は、ファイルアクセス要求に関するトークンをクライアントに発行することであり、どのファイルに対するどのタイプのトークンがどのクライアントに発行されたかを管理することである。クライアントは、それが適切なトークンを所有するまでファイルデータに対して必要なファイル操作を実行することはできない。

単一サイト UNIX ファイル共有セマンティクスを実現するためのトークンに基づく方法については、例を使って説明する(図9.21を参照)。簡単のため、この例ではすべてのファイルアクセス操作のタイプについて1つのタイプのトークンしかないと仮定する。

図9.21(a)は、サーバと2つのクライアントの初期状態を示している。このとき、クライアントAはファイル F_1 をアクセスするために、サーバに要求を送る。サーバは、ファイル F_1 のトークンがどこか他のクライアントに既に与えられていないかどうか状態情報を検査する。もし、他のクライアントに与え

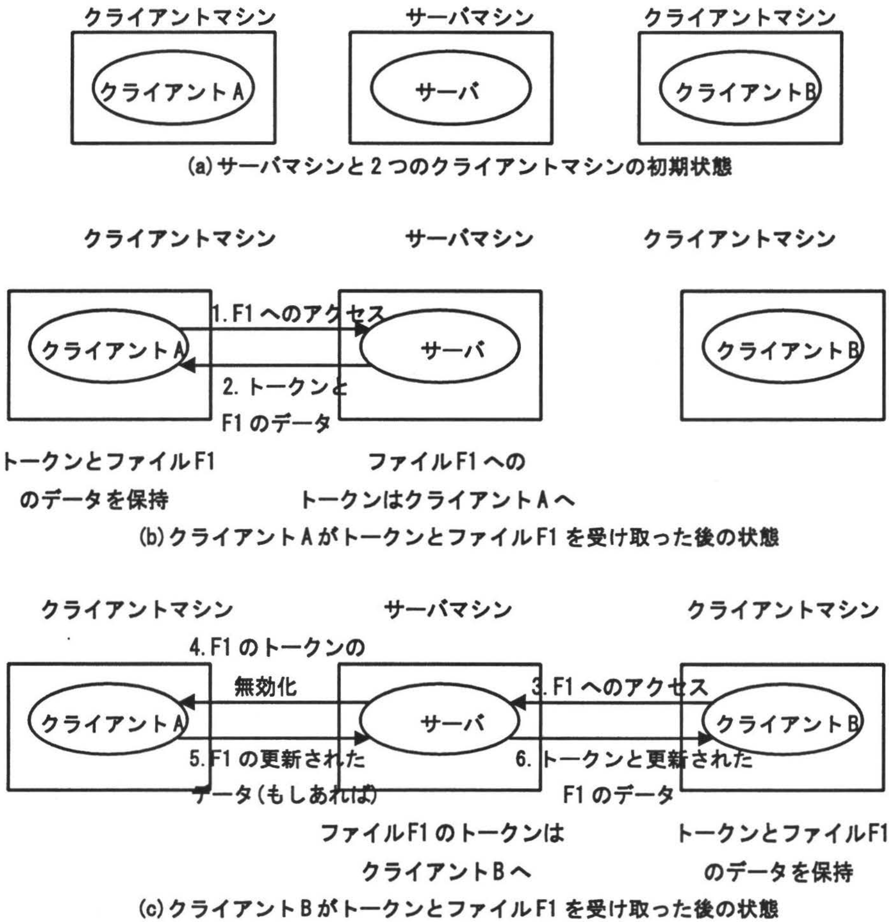


図 9.21 UNIX ファイル共有セマンティクスを実現するためのトークンに基づく方法

られていなければ、トークンと F_1 のデータがクライアントに A に送られ、この情報が将来の利用のために記録される。クライアント A は受信したファイルデータをキャッシュし、そしてこのデータに対して必要なだけのファイルアクセス操作を連続して実行する。図 9.21(b) は、クライアント A がトークンとファイル F_1 のデータを受信した後の、サーバとクライアントの状態を示している。

さて、幾らかの時間が過ぎた後で、クライアント B がサーバに対して同じファイル F_1 のアクセスを要求したと仮定する。サーバはその状態情報を検査して、F_1 に対するトークンがクライアント A に与えられていることを見つける。したがって、ファイル F_1 に対するトークンとデータをクライアント B にすぐに送ることはできない。最初に、ファイル F_1 に対するトークンを戻すようにクライアント A にメッセージを送る。開放メッセージを受けたクライアント A は、ファイル F_1 の更新されたデータ (もし更新が行われていれば) と一緒にトークンを返す。そして、ファイル F_1 のキャッシュデータを無効化する。サーバは、それから (もし必要なら) ファイル F_1 のローカルコピーを更新して、トークンと更新されたファイル F_1 のデータとをクライアント B へ送る。クライアント B では、受信したデータをキャッシュして、このデータに対するファイルアクセス操作を必要な回数だけ連続して実行する。図 9.21(c) は、クライアント B がトークンとファイル F_1 のデータを受信した後のサーバとクライアントのマシンの状態を示している。この方法では、単一システム UNIX ファイル共有セマンティクスが達成されている。なぜなら、サーバはトークンを同時に 1 つのクライアントにしか渡していないからである。そのトークンを所有しているクライアントは、そのファイルに対する排他的なアクセスが保証される。

性能を向上させるために、以下の技術を使ってより高い並列性を達成できる。

1. **タイプ別のトークン**:同じファイルを読み込むが決して書き込まない複数のクライアントでは、矛盾は生じない。したがって、クライアントが読み込み専用モードでファイルをアクセスするとき、他のクライアントが読み込み専用モードでそのファイルをアクセスしようとするのに、そのファイルのトークンを渡さない理由はない。したがって、ファイルのすべてのタイプの操作に 1 つのトークンを使う代わりに、タイプに応じたトークンが使われる。ファイルに対するオープン、読み込み、書き込み、ロック、ファイル状態の検査、そしてファイル状態の更新についての操作に分かれたトークンが存在する。サーバは、トークンの互換性に関する規則を知っている。つまり、ファイルデータへの読み込みアクセスに対して複数の読み込みトークンが発行されるが、ある

トークンに書き込みトークンが既に送られたファイルデータに対しては読み込みトークンも書き込みトークンも発行されない。

2. **細粒度トークン**:偽共有の問題を最小化するために、矛盾する操作に対応するトークンはファイル全体ではなくてファイルの一部のみを参照するようにする。結局、オープン、ファイル状態の検査、そしてファイル状態の更新の操作に対応するトークンはファイル全体に適用されるが、読み込み、書き込み、そしてロック操作に対応するトークンはファイルの一部についてのみ適用される。

各トークンは、2分の終了時間がある。したがって、もしクライアントが(クラッシュや他の理由によって)サーバからのトークン開放メッセージに対する応答を返さない場合、サーバはただ2分間待って、そしてトークンがクライアントから返却されたとみなして動作する。

NFS におけるファイル共有セマンティクスについてみると、NFS はファイル共有に UNIX セマンティクスをサポートしていない。あいまいなファイル共有セマンティクスを持っていることに対して、多くの批判を受けてきている。このセマンティクスでは、クライアントでキャッシュされたファイルのデータに対して書き込みを行ったとき、他のクライアントがそのファイルを読んだときに、タイミングによってその結果が見えたり見えなかったりする。なぜなら、NFS においては、各キャッシュデータブロックは、それぞれ3秒のタイマを持っている。キャッシュされたデータブロックは、そのタイマが終了した時点で無視される。更に、キャッシュされたファイルがオープンされたとき、クライアントはサーバにそのファイルがいつ最後に更新されたかを問い合わせる。ファイルがクライアントによってキャッシュされた後で最後の変更が行われているとき、そのキャッシュより古いコピーを無視して、新しいコピーをファイルサーバから取ってきて、このコピーをキャッシュする。最後に、NFS は30秒間隔で周期的書き込み変更伝搬方式を使っている。つまり、30秒に1回、キャッシュに対するすべての変更がサーバに対して送信される。キャッシュされた内容の無効化と変更の伝搬についてタイマに基づく方法を使っているために、NFS はデータキャッシュをうまく利用できない。

NFS とここで詳しく述べた DFS との重要な違いは、NFS は各マシンでサーバとクライアントの両者が同時に動作できる。つまり、どのクライアントもサーバになることもでき、そしてどのサーバもクライアントになることができる。すべてのマシンは独立であり、異なる管理環境に存在している。しかし、DFS では、サーバはクライアントが同じセル内にあることに依存している。し

たがって、DFS におけるサーバ-クライアント関係は、NFS よりもマスタ-スレーブにより近い。

9.11.5　DFS におけるファイルキャッシュ方式

本書では、既に DFS において最近アクセスされたファイルデータが、クライアントマシンにあるキャッシュ管理者によってキャッシュされることを説明した。クライアントマシンのローカルディスクは、この目的に使われる。しかし、ディスクレスクライアントマシンでは、ファイルデータのキャッシュのために主記憶が使われる。

図 9.21 に示すように、DFS ではキャッシュされたファイルデータに対する変更は、クライアントがファイルデータに対するトークン開放要求を受け取ったときにのみ伝搬される。同じ事がキャッシュの検証についてもいえる。つまり、クライアントマシンにキャッシュされたデータは、クライアントがファイルサーバからファイルデータの開放メッセージを受けるとその時点でのみ無効化 (そのキャッシュの内容が無用になる) される。クライアントがファイルデータに対して指定された操作のためのトークンを持っている限り、キャッシュされたデータは正しく、そしてクライアントはそれに対する操作を継続することができる。結果的に、キャッシュ検証に使われているこの方法は、サーバ主導方式という。

NFS で使われている変更の伝搬とキャッシュ検証の方式は、前の章で既に説明している。

9.11.6　DFS における複製とクローン

分散ファイルサービスは、ファイルを複数のファイルサーバに複製する機能を提供する。複製の単位はファイルセットである。つまり、ファイルセットのすべてのファイルは共に複製される。

1 つのファイルに複数の複製があることは、通常の利用者 (クライアントアプリケーション) には見えない。つまり、ファイル名はそのファイルの複製を持っているすべてのファイルサーバにマップされる。したがって、与えられたファイル名に対して、ファイルセット特定サーバはそのファイルが含まれるファイルセットを持っているファイルサーバのすべてのアドレスを返す。DFS は、複製制御において明示的な複製機構を使っている。つまり、1 つのファイルセットにおける複製の数とその位置はシステム管理者によって決定される。ファイ

ルセットサーバはファイルセット全体を複製するための1つのコマンドを持っている。

サーバマシンの複製サーバは、ファイルセットの複製の一貫性を維持する責任を持っている。この目的には、主コピープロトコルが用いられている。つまり、各複製されたファイルセットに対して、1つのコピーが主コピーとして指定され、すべての他のコピーは従コピーとする。複製されたファイルセットに対する読み込み操作は、主でも従でも、どのファイルセットのコピーを使っても実行できる。しかし、すべてのファイルセット上のファイルへの更新操作は、ファイルセットの主コピーでのみ直接実行される。主コピーの複製サーバは、定期的に従コピーの複製サーバに対して変更されたファイルの更新された版を送信する。そして、従コピーの複製サーバは自身の複製を更新する。

NFS は複数のサーバ上にファイルを複製する機能を提供しない。

ファイルセットの複製を可能とすることに加えて、DFS はファイルセットのクローンを作成する機能を提供する。この機能は、ファイルセットの仮想的な新しいコピーを他のディスクパーティション上に作成することを可能とし、そして古いコピーを読み込み専用とする。この機能は、ファイルセットの古い版を管理して、システム管理者が不注意によるファイルの削除から古い版を回復するために使うことができる。例えば、システム管理者はシステムに対して毎日真夜中にファイルセットのクローンを作成しておくように設定しておくことで、常にすべてのファイルの古い版という形で前日の作業が完全に残される。もし、利用者が不注意でファイルを削除した場合、彼又は彼女は常にファイルの古い版を得ることができ、そして今日の更新作業を再実行することができる。

ファイルセットのクローンは、多くの時間がかかることはない。なぜなら、ファイルセットの仮想的なコピーが行われるだけだからである。これは、つまりファイルセットのファイルのデータ構造だけが新しいパーティションにコピーされ、ファイルデータ自体はコピーされないからである。元のパーティションの古いデータ構造は読み込み専用になる。したがって、両方のデータ構造の集合が同じデータブロックを指している。新しいファイルセットにあるファイルが更新されると、新しいデータブロックが確保され、そのデータの更新された版が記述される。そして、新しいパーティションにおけるファイルデータに対応するデータ構造が新しいデータブロックを指すように更新される。元々のファイルセットにおけるファイルに対する更新要求は、誤りメッセージとともに拒絶される。

9.11.7　耐障害性

ファイルセットの複製を可能とするのに加えて、耐障害性を向上させることに役立つDFSのもう1つの重要な機能は、ファイル更新を回復可能な方法で記録するための先書き込みログ方式の利用である。DFSでは、ファイルに対して行われる更新毎にログがディスクに書き込まれる。ログの内容は、ファイルが変更された部分の古い値と新しい値を含む。システムがクラッシュから再起動したとき、ログはどの変更がファイルに対して既に実行されたか、そしてどの変更がまだ行われていないのかを検査するのに使う。このまだ行われていない変更が、システムクラッシュによって失われてしまった部分である。これらの変更は、ここで実行され、ファイルは一貫性のある状態に戻される。クラッシュが更新のためのログをとる前に起こり、1つの更新が失われた場合、失われた更新はその更新がそのファイルに対しては決して起こらなかったかのように扱われるため、一貫性を損なうことがない。したがって、ファイルは回復の後でも一貫性を保っている。

DFSにおけるログに基づくクラッシュ回復方式では、回復時間はログの長さに比例し、そしてディスクパーティションのサイズに独立である。これは、UNIXのような従来のシステムより高速な回復が可能となる。UNIXシステムでは、ファイルシステムの回復時間はそのディスクパーティションのサイズに比例している。

耐障害性について、NFSで使われている主な方法は、ステートレスサーバを使うことである。図9.21から、DFSサーバはステートフルである。なぜなら、クライアントに提供されたトークンを管理しておく必要があるからである。

9.11.8　原子トランザクション

DCEはトランザクション処理機能を提供しておらず、DFSの一部か独立した部品として提供される。これは、DCEが現在ミッションクリティカルな、分散**オンライントランザクション処理(OLTP)**アプリケーションを開発し実行するために必要なサービスを持っていないことによる。例えば、OLTPアプリケーションは、保証されたデータ一貫性、簡単なトランザクションセマンティクスを持つアプリケーションプログラミングインタフェース、そして共通のタスクをネットワークを越えて実行することで、複数のプロセスの共同作業を可能とするRPCをサポートするようにプログラムを拡張できることが必要

である。このようなサービスは現在DCEではサポートされていない。しかし、トランザクション処理機能を必要としているDCEの利用者は、Transarc CorporationのEncinaというOLTP技術が利用できる。**Encina**は、DCEの枠組みを拡張し、データ一貫性を保証した高信頼の分散OLTPシステムの構築を容易にするための、標準に基づいた分散サービス群を提供している。特に、分散OLTPのためにEncinaが提供するサービスは、トランザクション的な2フェーズコミットプロトコルを使った完全なデータ一貫性、単純化されたトランザクションセマンティクスによる高レベルなアプリケーションプログラミングインタフェース、そしてRPCを使った決定的な実行結果を達成するために必要な付加的なトランザクション上のセマンティクスである。

Encinaに加えて、他のよく知られているトランザクション環境は、**カスタマ情報制御システム (Customer Information Control System: CICS)** と **情報管理システム (Information Management System: IMS)** の2つで、共にIBMのシステムである。CICSは既に90か国以上、20,000以上の顧客に使われている。EncinaはIBMのCICSとの相互運用性を提供している。したがって、CICSはDCEとEncina技術の上に実装することができる。

9.11.9 DFSの利用者インタフェース

分散ファイルサービスは、様々なタイプの利用者に対して以下のようなタイプの利用者インタフェースをサポートする。

1. ファイルサービスインタフェース。DFSは、ディレクトリやファイル操作について、本来のオペレーティングシステムが持つコマンドを使っているため、利用者は新しいコマンドを覚える必要はない。例えば、UNIXシステムの利用者は、ディレクトリを変更するのにcdを使い、ディレクトリの内容をリストとするのにlsを使い、新しいディレクトリを作るのにmkdirを使うなどである。DFSは、それ自身の持つファイルシステム(Episode)でのみ動作する幾つかのコマンドも持っている。これらは、ファイルセットの割り当ての検査をするコマンドや、ファイルのサーバを特定するコマンドなどである。ファイルにアクセスするには、クライアントはファイルをグローバル名かそのセルの相対名によって指定する。DCEのオブジェクト名前機構の詳細は、第10章に示す。

2. アプリケーションプログラミングインタフェース。DFSのアプリケーショ

ンプログラミングインタフェースは、UNIX に非常に近い。そのため、アプリケーションプログラマは、ファイルを開くのに fopen()、ファイルを読み込むのに fread()、ファイルへの書き込みに fwrite()、ファイルを閉じるのに fclose() などといった標準ファイルシステム呼び出しを使うことができる。実際、ほとんどの既存のソフトウェアは、DFS ライブラリを使ってただリコンパイルするだけですぐ動作する。

3. 管理用インタフェース。DFS の管理用インタフェースは、ファイルセットを扱ったり、DFS ファイルサーバをインストールしたり削除したり、そしてファイルやディレクトリに関する ACL を操作するためのシステム管理者用のコマンドが提供されている。ファイルセットを扱うコマンドは、ファイルセットの作成、削除、移動、複製、クローン、バックアップ、又は再格納のためにシステム管理者によって利用される。例えば、システム管理者は、セル内にあるすべてのファイルサーバマシンの負荷のバランスをとるために、ファイルセットをあるサーバマシンから他のマシンに移動させることもある。

他方、ファイルサーバをインストールしたり削除したりするコマンドによって、システム管理者は動的にシステムを必要性の変化に応じて再構成することができる。例えば、システム管理者は、多すぎるクライアントと少しのサーバしか存在しないためにセル内の DFS の性能が良くないと通知される場合がある。この場合、彼又は彼女は新しいファイルサーバをセル内にインストールして、そして幾らかのファイルセットを既にセル内で存在しているファイルサーバからその新しいサーバに移動することができる。

最後に、ACL を操作するコマンドは、既に何人かの利用者に与えられていた許可を取り消したり、何人かの利用者に追加の許可を与えたりするためにシステム管理者によって利用される。

9.12　まとめ

ファイルとは、明示的な作成によって存在する名前を付けられたオブジェクトであり、システムの一時的な障害に対しては耐性を持ち、明示的な消去が行われるまで永続的なものである。ファイルシステムとは、オペレーティングシステムのサブシステムであり、ファイルの組織化、格納、読み出し、名前付け、共有、そして保護のようなファイル管理を実行する。分散ファイルシステムは、

旧来のファイルのタイムシェアリングモデルを分散して実現したものである。

分散ファイルシステムに求められる良い機能とは、透過性 (構造透過性、アクセス透過性、名前透過性、そして複製透過性)、利用者移動性、性能、利用の簡潔性と容易性、拡張性、高可用性、高信頼性、データ一貫性、安全性、そして異機種性である。

構造の観点から、ファイルは2つのタイプからなる。すなわち、非構造化と構造化ファイルである。他方、変更可能性の評価尺度に関して、変更可能あるいは不変ファイルという場合がある。

遠隔のファイルをアクセスするための2つの相補的なモデルは、遠隔サービスモデルとデータキャッシュモデルである。データキャッシュモデルを使ったファイルシステムにおいて、重要な設計項目はデータ転送の単位である。この目的で4つの広く使われている単位は、ファイル、ブロック、バイト、そしてレコードである。

共有ファイルでは、ファイル共有セマンティクスによって、いつ利用者から行われたファイルに対する変更を他の利用者からも見えるようにするかが定義される。4つの広く使われているファイル共有のセマンティクスは、UNIXセマンティクス、セッションセマンティクス、不変共有ファイルセマンティクス、そしてトランザクション型セマンティクスである。

今日、重要な利用が行われている分散ファイルシステムでは、幾つかの形式のファイルキャッシュが行われている。それは、性能の向上に加えて、その拡張性や信頼性にも寄与しているからである。分散ファイルシステムにおいて、キャッシュはサーバの主記憶、クライアントのディスク、あるいはクライアントと主記憶に置かれる。複数のクライアントノードにキャッシュされたファイルのデータの一貫性を維持することは、クライアントキャッシュを用いた分散ファイルシステムにおける重要な設計項目である。この課題を扱う方法は、キャッシュされたデータへの変更を、対応するファイルサーバへ伝搬する方式と、キャッシュされたデータの正しさを検証する方式に依存する。書き込みスルー方式と遅延書き込み方式は、変更伝搬のために広く使われている2つの方法である。キャッシュ検証方式は、クライアント主導方式とサーバ主導方式がある。

複製されたファイルとは、複数のコピーを持つファイルであり、それらのコピーが分離したファイルサーバに位置することである。複製ファイルを構成するコピーの集合のうちの各1つのコピーは、ファイルの複製と呼ばれる。分散システムにおいてファイルの複製を作ることで、以下の潜在的な利点が得られる。可用性の向上、信頼性の向上、応答時間の改善、ネットワークトラフィッ

クの軽減、システムスループットの改善、拡張性の向上、そして自律的な操作である。複製されたファイルが更新されたときのコピー間の一貫性を保つことは、ファイルの複製をサポートするファイルシステムにとって主要な設計項目である。この問題を扱うための広く使われている方式には、読み込み専用複製、任意読み込み全書き込みプロトコル、利用可能コピープロトコル、主コピープロトコル、そして定数に基づくプロトコルなどがある。

分散ファイルシステムの設計では、フォールトトレラント性は重要な設計項目である。障害に耐える分散システムの能力に直接影響する3つの主なファイルの特性は、可用性、堅固さ、そして回復可能性である。複製は、ファイルの可用性を向上させる主な機構である。堅固なファイルは、安定記憶のような冗長化技術によって実現される。回復可能なファイルは、原子的な更新方式によって実現されている。

サーバは、ステートフル、あるいはステートレスのサービスパラダイムのどちらかを使って実装される。ステートフルファイルサーバは、1つのアクセスから次のアクセスまで、クライアントの状態情報を保持している。他方で、情態なしファイルサーバは、クライアントの情態情報は一切管理しない。したがって、クライアントからの各要求は、必要な操作を完全に実行できるだけの必要なすべてのパラメータを持つ必要がある。ステートレスサービスパラダイムは、サーバによってクライアントの状態情報を保持していないため、そして各要求にすべての必要な情報が含まれているため、クラッシュからの回復が容易である。

原子トランザクション(あるいは短縮して単なるトランザクション)は、障害が存在する並行計算において、分割できない操作の集まりからなる計算である。トランザクションの3つの基本的な性質は、原子性、逐次可能性、そして不変性である。分散トランザクションサービスは、従来のトランザクションサービスの拡張であり、2つ以上のサーバによって管理されているファイルを含むトランザクションをサポートできる。入れ子トランザクションは、従来のトランザクションの一般化であり、1つのトランザクションは下位トランザクションと呼ばれる他のトランザクションから構成されてもよい。

分散ファイルシステムの設計のための基本原則は、Satyanarayananが提案したように、クライアントのもつ燃焼のサイクルを知ること、可能な場合は必ずキャッシュすること、利用の特性を知ること、システム全体に渡る知識と変更を最小化すること、可能な限り最小の実体についてのみ保証すること、そして可能であればバッチとすることである。

9.13 演習問題

9.1 集中タイムシェアリングシステムのためのファイルシステムと、分散ファイルシステムの設計ではどんな側面が異なっているか。

9.2 分散ファイルシステムの主な要素を上げよ。分散ファイルシステムの機能がこれらの要素に分割されている理由は何か。

9.3 分散ファイルシステムの設計において、高い可用性と高い拡張性は相互に関連している性質である。議論せよ。

9.4 分散ファイルシステムの設計において、高い性能と高い信頼性は矛盾する性質である。議論せよ。

9.5 不変ファイルとは何か。不変ファイルのみを用いて正しく機能するファイルシステムは設計できるか。もし可能なら、どのように基本ファイル操作 (create, read, write, delete) がこのファイルシステムにおいて共有されたファイルに実行されるかを説明せよ。

9.6 分散ファイルシステムのデータキャッシング機構に対して、全ファイルキャッシュとブロックキャッシュモデルを使った場合の相対的な利点と欠点を議論せよ。

9.7 ファイルアクセスにデータキャッシュモデルを使う分散ファイルシステムに対して、利用できる4つのデータ転送モデルについて、どのモデルが以下のタイプの分散システムに適しているか。

(a) ディスクレスワークステーションをサポートしている分散システム

(b) 各ノードが大きなディスク記憶を持つ分散システム

(c) 各ファイルがレコードの集まりという構造化されたファイルモデルを使っている分散システム

もし、あるケースに2つ以上のモデルが適している場合、読者はどちらのモデルが良いと思うか。その理由を示せ。

9.8 読者の意見では、以下のタイプの分散システムにおけるキャッシュデータをどこに (サーバの主記憶、クライアントのディスク、クライアントの主記憶) 配置するのが良いか (解答に対する理由も示せ)。

(a) ディスクレスワークステーションをサポートする。

(b) データアクセス単位において、ファイルレベル転送モデルを用いている。

(c) セッションセマンティクスを使っている。

(d) コネクションの切断状態での操作をサポートするように設計されている。

(e) ファイルサーバに対するクライアントの数の割合が非常に大きいシス

テム。

(f) 大きいファイルを扱う必要のあるシステム。

(g)UNIX 的なファイル共有セマンティクスをサポートするシステム。

ある特定のケースにおいて、複数の位置が適している場合、読者はどこの位置が良いと思うか。その理由を述べよ。

9.9 読者は、セッションセマンティクスをサポートする必要のある分散ファイルシステムのキャッシュ方式を設計すると仮定する。読者の設計において、以下の問題のそれぞれについて、適切な解決策を提案せよ。

(a) キャッシュはどこにおくか。

(b) データキャッシュの単位は何にするか。

(c) キャッシュデータへの変更を、いつマスターのコピーに反映するか。

(d) キャッシュのコピーがそのマスターコピーと同一であるかどうかをいつ検査するか。

読者の解答についての理由も述べよ。

9.10 ある分散オペレーティングシステム設計者は、オブジェクトの複製とキャッシュは多かれ少なかれ同様の利点を分散システムに提供しているので、同じ分散システムに 2 つの考えを共に実現する必要はないという。彼又は彼女の意見は正しいか。読者の意見の理由を述べよ。そして、以下のタイプの分散オペレーティングシステムについて、相対的な利点と欠点を述べ、両者の違いを説明せよ。

(a) オブジェクトキャッシュは実現しているが、オブジェクト複製は実現していない。

(b) オブジェクト複製は実現しているが、オブジェクトキャッシュは実現していない。

(c) オブジェクトキャッシュとオブジェクト複製の両者を実現している。

9.11 分散オペレーティングシステムの設計において、データキャッシュ機構は、多くの異なるタイプのデータをキャッシュするのに使われる。各データのタイプ毎に、別のキャッシュが管理される。読者の意見では、キャッシュされたデータは常に最新のものにしておく必要があるか。もしそうなら、なぜかを説明せよ。もしそうでないなら、常に最新でないかもしれないキャッシュをプロセスがアクセスしたときでも正しく機能するシステムの例を与えよ。

9.12 あるファイルシステムは、クライアントキャッシュの内容を検査するために、クライアント主導方式を使うと仮定する。正しさの検査は、データのキャッシュ版が最後に変更された時間と、サーバのマスターコピー版とを比較

することで実施すると仮定する。このファイルシステムは、すべてのノードのクロックが同期していなければ正しく動作しない。すべてのノードのクロックが同期していないシステムのキャッシュの検査のためにクライアント主導の方式を使うための方法を提案せよ。

9.13 分散ファイルシステムの以下の性質について違いを述べよ。

(a) 可用性の高さ

(b) 回復性の高さ

(c) 堅固性の高さ

各これらの性質を実現するために用いられる適切な機構を上げよ。

9.14 安定記憶システムは、どのように障害が発生しうるディスクを信頼性の高い装置に変換しているかを説明せよ。この章の議論において、2つの既存のディスクを用いて設計された安定記憶システムを見てきた。この考えは、よりよいフォールトトレラント性能のために3つのディスクに拡張できるか。もしできないなら、理由を述べよ。もし可能なら、どのように拡張するかを述べよ。

9.15 状態を持つファイルサーバは、そのクライアントの状態情報を記録している。このタイプのファイルサーバに関係する問題点は何か。状態を持つファイルサーバが使うことが必要となる例を2つ上げよ。

9.16 あるワークステーション-サーバモデルに基づく分散システムは、manコマンドを使ってオンラインマニュアルを利用者が読むことを可能とすることで、オンラインヘルプ機能を提供している。マニュアルを格納するためのシステムにおける様々な位置を提案し、その相対的な利点と欠点を議論せよ。

9.17 トランザクションとは何か。トランザクションの原子性を脅かす2つの主な要因は何か。コミットとアボートの両者において、トランザクションの原子性はどのように保証されているか。

9.18 ファイルサービスにどうしてトランザクションが必要か。以下のことを実行する上で、トランザクションがどのように有用かを示す例を与えよ。

(a) 障害イベントが生じた際に、ファイルの回復性を向上する。

(b) 複数のクライアントによる統一された方法により変更可能ファイルの並行共有を可能とする。

9.19 トランザクションにおける逐次化可能性の必要性について議論せよ。この性質を満足するための機構を考案する上での主な目標を示せ。この性質を満足するトランザクション機能の実現に使うことができる少なくとも3つの機構を述べよ。読者が示した機構を比較し、主な目標にそれぞれがどれだけ近いかを示せ。

9.20 回復可能な方法でファイルへの更新を記録するためのファイルシステムの設計において利用できる 2 つの方法を示せ。これは、利用者に対して、open から close までのセッションにおいて加えられたファイルへの更新を取り消し、ファイルをそのセッションが始まる前の状態に戻すための自由度を提供する。

9.21 図 9.9 の 2 つの銀行トランザクションにおける 3 つの口座の残高がすべての$100 であるとする。この 2 つのトランザクションに対して、z に異なる値を生じるすべてのスケジュールを数えよ。どのスケジュールが正当か。

9.22 あるアプリケーションは、以下に定義された 3 つのトランザクション T_1、T_2 そして T_3 からなる。

```
T1: begin_transaction
        read(x);read(z);write(x-5);write(z+5);
    end_transaction
T2: begin_transaction
        read(z);write(z-8);read(y);write(y+8);
    end_transaction
T3: begin_transaction
        read(x);write(x+4);read(y);write(y-4);
    end_transaction
```

これらの 3 つのトランザクションの並行性が以下を利用してどのように制御されるかを述べよ。

(a) 読み込みと書き込み操作に対して同じ種類のロックを使う

(b) 種類に応じたロックを使う

(c) 書き込み意志ロックを使う

(d) 楽観並行制御方式を使う

(e) タイムスタンプに基づく方式を使う

9.23 逐次的等価スケジュールとは何か。前の問題の 3 つのトランザクションに対して、順序的等価な少なくとも 6 つのスケジュールを示せ。

9.24 図 9.10 は、期待しない結果を生じる図 9.9 の 2 つのトランザクションの 2 つのスケジュールを示している。もし、2 つのトランザクションが 2 フェーズロックを用いれば、これらのスケジュールは正しい結果を出すことを示せ。

9.25 あるアプリケーションのすべてのトランザクションが 2 フェーズロックプロトコルを用いれば、その操作をインターリーブすることによるすべてのス

ケジュールは逐次化されることを証明せよ。

9.26 偽共有とは何か。偽共有問題と対抗するロックの粒度の重要性について議論せよ。

9.27 トランザクションデッドロックとは何か。トランザクションデッドロックがどのように発生するかを示す例を与えよ。

(a) 読み込みと書き込み操作において、同じ種類のロックが使われる場合。

(b) 種類に応じたロックが使われる場合。

トランザクションデッドロックを避けるために利用できる方法を示し、上の (a) と (b) で起こる読者が例示したデッドロックにその方法を適用せよ。

9.28 ファイルシステムのトランザクション機構の実現に楽観並行制御機構が用いられると仮定する。このシステムにおいて、2 つの並行実行の例を示せ。

(a) すべてのトランザクションについて、正当性検査が正常で、成功裏にコミットされたもの。

(b) 正当性検査に失敗して、トランザクションがアボートされたもの。

9.29 タイムスタンプに基づく並行制御方式が楽観並行制御方式より良いのは、どんなところか。前の問題の (b) の場合に対して読者が与えた例について、タイムスタンプに基づく並行制御方式を用いて、この場合アボートされたトランザクションがその第一フェーズを完了できないことを示せ。

9.30 分散システムにおいて、入れ子トランザクション方式が従来のトランザクション方式より利点があるのはどんな点か。以下の入れ子トランザクションファミリに対する質問に答えよ。

(a) ファミリ内の他のトランザクションに独立にトランザクションをコミットできるか。

(b) ファミリ内の他のトランザクションに独立にトランザクションをアボートできるか。

(c) あるトランザクションによってなされたデータ項目に対する変更は、ファミリ内の他のトランザクションに見えるようになるか。

(d) ファミリ内のトランザクションが失敗するとどうなるか。

(e) あるトランザクションによる更新はいつ永続化するか。

(f) トランザクションファミリの全体がコミットとして成功したことはいつクライアントに報告されるのか。

答えに対する理由を述べよ。

9.31 ワークステーション-サーバモデルに基づく分散システムにおいて、サーバマシンはファイルサーバとして専用に動作する。このファイルシステムは、

クライアントの要求を処理するために遠隔サービスモデルを使う。時間の経過により、システム内のワークステーションの数がだんだんと多くなり、ファイルサーバがクライアントからの要求によって過負荷になったためにシステム性能が下がってきたことが解った。分散オペレーティングシステムの専門家として、読者はこの問題を解決するように依頼されている。この問題を解決するために利用できる3つの異なる解決策を示せ。

9.32 2フェーズマルチサーバコミットプロトコルを記述せよ。以下に示した銀行トランザクションにおいて、XとYを2つの異なる銀行の支店にある口座とする。2つの支店のローカルな記録はサーバS1、S2にそれぞれ管理されている。

a_1: begin_transaction
a_2: 口座 X の残高 (x) を読み込む
a_3: 口座 Y の残高 (y) を読み込む
a_4: $(x-5)$ を口座 X に書き込む
a_5: $(y+5)$ を口座 Y に書き込む
a_6: end_transaction

上のトランザクションに対して、S1は調停者、S2は作業者の場合、このトランザクションの正常実行の場合のクライアントとファイルサーバS1、S2との交換されるメッセージのリストを示せ。もし、a_1 の操作を実行中にS2が故障した場合、何が起こるか。

参考文献

[Agrawal and Jalote 1995] Agrawal, G., and Jalote, P., "Codin g-Based Replication Schemes for Distributed Systems," IEEE Transactions on Parallel and Distributed Systems, Vol. 6, No. 3, pp. 240-251 (1995).

[Ahamad et al. 1991] Ahamad, M., Ammar, M. H., and Cheung, S. Y., "Multi-dimensional Voting," ACM Transactions on Computer Systems, pp. 399-431 (November 1991).

[Barghouti and Kaiser 1991] Barghouti, N. S., and Kaiser, G. E., "Concurrency Control in Advanced Database Applications," ACM Computing Surveys, Vol. 23, No. 3, pp. 269-318 (1991).

[Bernstein and Goodman 1981] Bernstein, P. A., and Goodman, N., "Concurrency Control in Distributed Database Systems," ACM Computing Surveys, Vol. 13, No. 2, pp. 185-221 (1981).

[Bernstein and Goodman 1984] Bernstein, P. A., and Goodman, N., "An Algorithm for Concurrency Control and Recovery in Reeplicated Distributed Databases," ACM Transactions on Database Systems, Vol. 9, No. 4, pp. 596-615 (1984).

[Bernstein et al. 1980] Bernstein, P. A., Shipman, D. W., and Rothnie, J. B., "Concurrency Control in a System for Distributed Databases (SDD-1)," ACM Transactions on Database Systems, Vol. 5, No. 1, pp. 18-51 (1980).

[Bernstein et al. 1987] Bernstein, P. A., Hadzilacos, V., and Goodman, N., Concurrency and Recovery in Database Systems, Addison-Wesley, Reading, MA, pp. 289-307 (1987).

[Brown et al. 1985] Brown, M., Kolling, K., and Taft, E., "The Alpine File System," ACM Transactions on Computer Systems, Vol. 3, No. 4, pp. 261-293 (1985).

[Chen et al. 1995] Chen, K., Bunt, R. B., and Eager, D. L., "Write Caching in Distributd File Systems," In: Proceedings of the 15th IEEE International Conference on Distributed Computing Systems, IEEE, New York, NY (May-June 1995).

[Chutani et al. 1992] Chutani, S., Anderson, O. T., Kazar, M. L., Leverett, B. W., Mason, W. A., and Sidebotham, R. N., "The Episode File System," In: Proceedings of the 1992 USENIX Winter Conference. USENIX Association, Berkeley, CA, pp. 43-60 (1992).

[Claybrook 1992] Claybrook, B., OLTP: Online Transaction Processing Systems, John Wiley, New York (1992).

[Coulouris et al. 1994] Coulouris, G. G., Dollimore, J., and Kindberg, T., Distributed Systems Concepts and Design, 2nd ed., Addison-Wesley, Reading, MA (1994).

[Davcev and Burkhard 1985] Davcev, D., and Burkhard, W. A., "Consistency and Recovery Control for Replicated Files," In: Proceedings of the 10th ACM Symposium on Operating Systesms Principles, Association for Computing Machinery, New York, NY, pp. 87-96 (December 1985).

[Dion 1980] Dion, J., "The Cambridge File Server," ACM Operating Systems Review, Vol. 14, No. 4, pp. 26-35 (1980).

[Eswaran et al. 1976] Eswaran, K. P., Gray, J. N., Lorie, R. A., and Traiger, I. L., "The Notions of consistency and Predicate Locks in

a Database System," Communications of the ACM, Vol. 19, No. 11, pp.624-633 (1976).

[Fridrich and Older 1985] Fridrich, M., and Older, W., "Helix: The Architecture of the XMS Distributed File System," IEEE Computer, pp.21-29 (1985).

[Gifford 1979a] Gifford, D. K., "Weighted Voting for Replicated Data," In: Proceedings of the 7th ACM Symposium on Operating Systems Principles, Association for Computing Machinery, New York, NY, pp. 150-159 (December 1979).

[Gifford 1979b] Gifford, D. K., "Violet: An Experimental Decentralized System," ACM Operating Systems Review, Vol. 13, No. 5 (1979).

[Gifford et al. 1988] Gifford, D. K., Needham, R. M., and Schroeder, M. D., "The Cedar File System," Communications of the ACM, Vol. 31, No. 3, pp. 288-298 (1988).

[Goscinski 1991] Gscinski, A., Distributed Operating Systems, The Logical Design, Addison-Wesley, Reading, MA (1991).

[Gray 1978] Gray, J. N., "Notes on Database Operating Systems," Lecture Notes in Computer Science, Vol. 60, Springer-Verlag, Berlin, pp. 393-481 (1978).

[Gray and Cheriton 1989] Gray, C., and Cheriton, D., "Leases: An Efficient Fault-Tolerant Mechanism for Distributed File System Consistency," In: Proceedings of the 11th ACM Symposium on Operating Systems principles, Association for Computing Machinery, New York, NY, pp. 202-210 (1989).

[Gray and Reuter 1993] Gray, J., and Reuter, A., Transaction Processing: Concepts and Techniques, Morgan Kaufmann, San Francisco, CA (1993).

[Gray et al. 1981] Gray, J. N., McJones, P., Blasgen, M. W., Lorie, R. A., Price, T. G., Putzulu, G. F., and Traiger, I. L., "The Recovery Manager of the System R Database Manager," ACM Computing Surveys, Vol. 13, No. 2, pp. 223-242 (1981).

[Harder and Reuter 1983] Harder, T., and Reuter, A., "Principles of Transaction- Oriented Database Recovery," Computing Surveys, Vol. 15, No.4 (1983).

[Herlihy 1986] Herlihy, M., "A Quorum-Consensus Replication Method for Abstract Data Types," ACM Transactions on Computer Systems, Vol.4, No.1, pp.32-53 (1986).

[Howard et al. 1988] Howard, J. H., Kazar, M. L., Menees, S., G., Nichols, D. A., Satyanarayanan, M., Sidebotham, R. N., and West, M. J., "Scale and Performance in a Distributed File System," ACM Transactions on Computer Systems, Vol. 6, No. 1, pp. 51-81 (1988).

[Israel et al. 1978] Israel, J. E., Mitchell, J. G., and Sturgis, H. E., "Separating Data From Function in a Distributed File System," In: D. Lanciaux (Ed.), Operating Systems: Theory and Practice, North Holland, Amsterdam, pp. 17-22 (1978).

[Jalote 1994] Jalote, P., Fault Tolerancein Distributed Systems, Prentice-Hall, Englewood Cliffs, NJ (1994).

[Kazar et al. 1990] Kazar, M. L., Leverett, B. W., Anderson, O. T., Apostolides, V., Bottos, B. A., Chutani, S., Everhart, C. F.l, Mason, W. A., Tu, S. T., and Zayas, E. R., "Decorum File System Architectural Overview," In: proceedings of the 1990 USENIX Summer Conference, USENIX Association, Berkeley, CA, pp. 151-163 (1990).

[Khanna 1994] Khanna, R. (Ed.), Distributed Computing: Implementation and Management Strategies, Prentice-Hall, Englewood Cliffs, NJ (1994).

[Kistler 1995] Kistler, J. J., Disconnected Operation In a Distributed File System, Springer-Verlag, New York, NY (1995).

[Kistler and Satyanarayanan 1992] Kistler, J. J., and Satyanarayanan, M., "Disconnected Operation in the Coda File System," ACM transactions on Computer Systems, Vol. 10 No. 1 (1992).

[Kotz and Ellis 1993] Kotz, D., and Ellis, C. S., "Caching and Writeback Policies in Parallel File Systems," Journal of Parallel and Distributed Computing, Vol. 17, Nos. 1 and 2, pp. 140-145 (1993).

[Kumar 1991] Kumar, A., "Hierarchical Quorum Consensus: A New Algorithm for Managing Replicated Data," IEEE Transactions on Computers, Vol. 40, No. 9, pp. 996-1004 (1991).

[Kung and Robinson 1981] Kung, H. T., and Robinson, J. T., "On Optimistic Methods for Concurrency Control," ACM Transactions on Database Systems, Vol. 6, No. 2, pp. 213-226 (1981).

[**Ladin et al. 1992**] Ladin, R., Liskov, B., Shrira, L., and Ghemawat, S., "Providing Availability Using Lazy Replication," ACM Transactions on Computer Systems, Vol. 10, No. 4, pp. 360-391 (1992).

[**Lampson 1981**] Lampson, B. W., "Atomic Transactions in Distributed Systems—Architecture and Implementation," In: B. W. Lampson, M. Paul, and H. J. Siegart (Eds.), An Advanced Course, Lecture Notes in Computer Science, Vol. 105, Springer-Verlag, New York, NY, pp. 246-264 (1981).

[**Lazowska et al. 1986**] Lazowska, E. D., Zahorjan, j., Cheriton, D., and Zwaenepoel, W., "File Access Performance of Diskless Workstations," ACM Transactions on Computer Systems, Vol. 4, No. 3, pp. 238-268 (1986).

[**Leach et al. 1983**] Leach, P. J., Levine, P. H., Douros, B. P., Hamilton, J. A., Nelson, D. L., and Stumpf, B. L.," The Architecture of an integrated Local Network," IEEE Journal on Selected Areas in Communication, Vol.SAC-1, No. 5, pp. 842-857 (1983).

[**Levine 1986**] Levine, P. H., "The Apollo DOMAIN Distributed File System," In Y. Parker et al., (Eds.), Distributed Operating Systems: Theory and Practice, NATO ASI Series, Springer-Verlag, New York, NY, Vol.F28, pp. 241-260 (1986).

[**Levy and Silberschatz 1990**] Levy, E., and Silberschatz, A., "Distributed File Systems: Concepts and Examples," ACM Computing Surveys, Vol.22, No. 4, pp.321-374 (1990).

[**Liskov et al. 1991**] Liskov, B., Ghemawat, S., Gruber, R., Johnson, P., Shrira, L., and Williams, M., "Replkication in the Harp File System," In: Proceedings of the 13th ACM Symposium on Operating Systems Principles, Association for Computing Machinery, New York, NY, pp. 226-238 (1991).

[**Lockhart Jr. 1994**] Lockhart, Jr., H. W., OSF DCE: Guide to Developing Distributed Applications, IEEE Computer Society Press, Los Alamitos, CA (1994).

[**Lyon et al. 1985**] Lyono, B., Sager, G., Chang, J. M., Goldberg, D., Kleiman, S., Lyon, T., Sandberg, R., Walsh. D., and Weiss, P., "Overview of the SUN Network File System," In: Proceedings of the USENIX Conference, USENIX Association, Berkeley, CA, pp. 1-8 (January 1985).

[McKusick et al. 1985] McKusick, M. K., Karels, M. J., and Leffler, S. J., "Performance Improvements and Functional Enhancements in 4.3BSD," In: Proceedings of the 1985 USENIX Summer Conference, USENIX Association, Berkeley, CA, pp. 519 531 (1985).

[Mitchell and Dion 1982] Mitchell, J. G., Dion, J., "A Comparison of Two Network-Based File Servers," Communications of the ACM, Vol. 25, No. 4, pp. 233-245 (1982).

[Morris et al. 1986] Morris, J. H., Satyanarayanan, M., Conner, M., H., Howard, J. H., Rosenthal, D. S. H., and Smith, F. D., "Andrew: A Distributed Personal Computing Environment," Communications of the ACM, Vol. 29, No. 3, pp. 184-201 (1986).

[Moss 1985] Moss, E., Nested Transactions, An Approach to Reliable Distributed Computing, MIT Press, Cambridge, MA (1985).

[Mueller et al. 1983] Mueller, E. T., Moore, J. D., and Popek, G. J., "A Nested Transaction Mechanism for Locus," In: Proceedings of the 9th ACM Symposium on Operating Systems Principles, Association for Computing Machinery, New York, NY, pp. 71-85 (1983).

[Mullender and Tanenbaum 1984] Mullender, S. J., and Tanenbaum, A. S., "Immediate Files," Software Practice and Experience, Vol. 14, No.4, pp.365-368 (1984).

[Mullender and Tanenbaum 1985] Mullender, S. J., and Tanenbaum, A. S., "A Distributed File Service Based on Optimistic Concurrency Control," In: Proceedings of the 10th ACM Symposium on Operating Systems Principles, Associatio for Computing Machinery, New York, NY, pp.51-62 (December 1985).

[Needham and Herbert 1982] Needham, R. M., and Herbert, J., The Cambridge Distributed Computing System, Addison-Wesley, Reading, MA (1982).

[Nelson et al. 1988] Nelson, M. N., Welch, B. B., and Ousterhout, J. K., "Caching in the Sprite Network File System," ACM Transactions on Computer Systems, Vol. 6, No. 1, pp. 134-154 (1988).

[Ousterhout et al. 1985] Ousterhout, J. K., Costa, D., Harrison, D., Kunze, J. A¿, Kupfler, M., and Thompson, J. G., "G Trace-Driven Analysis of the UNIX 4.2 BSD File System," In: Proceedings of the 10th ACM Symposium on Operating Systems Principles, Association for Computing Machinery, New York, NY, pp. 15-24 (December 1985).

[Popek and Walker 1985] Popek, G. J., and Walker, B. J., The LOCUS Distributed System Architecture, MIT Press, Cambridge, MA (1985).

[Ramamritham and Chrysanthis 1996] Ramamritham, K., and Chrysanthis, P. K. (Eds.), Advances in Concurrency Control and Transaction Processing, IEEE Computer Society Press, Los Alamitos, CA (1996).

[Rangarajan et al. 1995] Rangarajan, S., Setia, S., and Tripathi, S. K., "A Fault-Tolerant Algorithm for Replicated Data Management," IEEE Transactions on Parallel and Distributed Systems, Vol. 6, No. 12, pp. 1271-1282 (1995).

[Reed 1983] Reed, D. P., "Implementing Atomic Actions on Decentralized Data," ACM Transactions on Computer Systems, Vol. 1, No. 1, pp. 3-23 (1983).

[Rifkin et al. 1986] Rifkin, A. P., Forbes, M. P., Hamilton, R. L., Sabrio, M., Shar, S., and Yueh, K., "RFS Architectural Overview," In: Proceedings of the USENIX Conference, Atlanta, GA, USENIX Association, Berkeley, CA, pp. 248-259 (1986).

[Rosenberry et al. 1992] Rosenberry, W., Kenney, D., and Fisher, G., OSF DISTRIBUTED COMPUTING ENVIRONMENT, Understanding DCE, O'Reilly, Sebastopol, CA (1992).

[Sandberg 1987] Sandberg, R., "The Sun Network File System: Design, Implementation and Experience," In: Proceedings of the USENIX Summer Conference, USENIX Association, Berkeley, CA, pp. 300-314 (June 1987).

[Sandberg et al. 1985] Sandberg, R., Goldberg, D., Kleinman, S., Walsh, D., and Lyon, B., "Design and Implementation of the SUN Network File System," In: Proceedings of the USENIX Summer Conference, Portland, OR, USENIX Association, Berkeley, CA, pp. 119-130 (June 1985).

[Santifaller 1994] Santifaller, M., TCP/IP and ONC/NFS, Internetworking in a UNIX Environment, 2nd ed., Addison-Wesley, Reading, MA (1994).

[Satyuanarayanan 1990a] Satyanarayanan, M., "A Survey of Distributed File Systems," Annual Review of Computer Science, Vol. 4, pp. 73-104 (1990).

[Satyanarayanan 1990b] Satyanarayanan, M., "Scalable, Secure, and Highly Available Distributred File Access," IEEE Computer, Vol. 23, No. 5, pp. 9-21 (May 1990).

[Satyanarayanan 1992] Satyanarayanan, M., "The Influence of Scale on Distributed File System Design," IEEE Transactions on Software Engineering, Vol. 18, No. 1 (January 1992).

[Satyanarayanan 1993] Satyanarayanan, M., "Distributed File Systems," In: S. Mullender (Ed.), Distributed Systems, 2nd ed., Association for Computing Machinery, New York, NY, pp.353-383 (1993).

[Satyanarayanan et al. 1985] Satyanarayanan, M., Howard, J. H., Nichols, D. A., Sidebotham, R. N., Spector, A. Z., and West, M. J., "The ITC Distributed File System: Principles and Design," In: Proceedings of the 10th ACM Symposium on Operating Systems Principles, Association for Computing Machinery, New York, NY, pp. 35-50 (December 1985).

[Satyanarayanan et al. 1990] Satyanarayanan, M., Kistler, J. J., Kumara, P., Okasaki, M. E., Siegel, E. H., and Steere, D. C., "Coda: A Highly Available File System for a Distributed Workstation Environment," IEEE Transactions on Computers, Vol. 39, no. 4, pp. 447-459 (April 1990).

[Schroeder et al. 1985] Schroeder, M. D., Gifford, D. K., and Needham, R. M., "A Caching File System for a Programmer's Workstation," In: Proceedings of the 10th ACM Symposium on Operating Systems Principles, Association for Computing Machinery, New York, NY, pp. 25-34 (December 1985).

[Silberschatz and Galvin 1994] Silberschatz, A., and Galvin, P. B., Operating System Concepts, 4th ed., Addison-Wesley, Reading, MA (1994).

[Singhal and Shivaratri 1994] Singhal, M., and Shivaratri, N. G., Advanced Concepts in Operating Systems,k McGraw-Hill, New York (1994).

[Smith 1982] Smith, A. J., "Cache Memories," ACM Computing Surveys, Vol. 14 No. 3, pp. 473-530 (1982).

[Stalling 1995] Stalling, W., Operating Systems, 2nd ed., Prentice-Hall, Englewood Cliffs, NJ (1995).

[Sturgis et al. 1980] Sturgis, H., Mitchell, J. G., and Israel, J., "Issues in the Design and Use of a Distributed File System," ACM Operating Systems Review, Vol. 14, No. 3, pp. 55-69 (1980).

[Svobodova 1984] Svobodova, L., "File Servers for Network-Based Distributd Systems," ACM Computing Surveys, Vol. 16, No. 4, pp. 353-398 (1984).

[Tanenbaum 1987] Tanenbaum, A. S., Operating Systems: Design and Implementation, Prentice-Hall, Englewood Cliffs, NJ (1987).

[Tanenbaum 1995] Tanenbaum, A. S., Distributed operating Systems, Prentice-Hall, Englewood Cliffs, NJ (1995).

[Terry 1987] Terry, D. B., "Caching Hints in Distributed Systems," IEEE Transactions on Software Engineering, Vol. SE-13, No. 1, pp. 48-54 (1987).

[Tichy and Ruan 1984] Tichy, W. F., and Ruan, Z., "Towards a Distributed File System," In: Proceedings of the Summer USENIX Conference, USENIX Association, Berkeley, CA, pp. 87-97 (June 1984).

[Tomlinson et al. 1985] Tomlinson, G. M., Keeffe, D., Wand, I. C., and Wellings, A. J., "The PULSE Distributed File System," Software Practice and Experience, Vol. 15, No. 11, pp. 1087-1101

[Turek and Shasha 1992] Turek, J., and Shasha, D., "The Many Jaces of Consensus in Distributed Systems," IEEE Computer, Vol. 25, No. 6, pp. 8-17 (1992).

[Van Renesse and Tanenbaum 1988] Van Renesse, R., and Tanenbaum, A. S., "Voting with Ghosts," In: Proceedings of the 8th IEEE International Conferene on Distributed Computing Systems, IEEE Press, New York, NY. PP. 456-461 (June 1988).

[Weihl 1933] Weihl, W. E., "Transaction-Processing Techniques," In: S. Mullender (Ed.), Distributed Systems, 2nd ed., Association for Computing Machinery, New York, NY, pp. 329-352 (1993).

[Weikum 1991] Weikum, G., "Principles and Realization of Multilevel Transaction Management," ACM Transactions on Database Systems, Vol. 16, No. 1, pp. 132-140 (1991).

インターネット参考文献

分散ファイルシステムに関する参考文献は、以下のところで見つけることができる。

ftp:ftp.cs.umanitoba.ca/pub/bibliographies/Distributed/distfs.html
ftp:ftp.cs.umanitoba.ca/pub/bibliographies/Distributed/dist.sys.html

分散システムにおけるオブジェクト複製に関する参考文献は、以下のところで見つけることができる。

ftp:ftp.cs.umanitoba.ca/pub/bibliographies/Distributed/Dist.Sys.html

AFS(Andrew File System) に関する参考文献は、以下のところで見つけることができる。

http://www.transarc.com/Product/AFS/FAQ/faq.html#sub6

第10章

名前付け

10.1 はじめに

分散システムは、プロセス、ファイル、I/O装置、メールボックス及びノードといったいくつかのタイプのオブジェクトをサポートする。分散システムの**名前付け機能 (naming facility)** により、ユーザ及びプログラムがオブジェクトに文字列の名前を割り当てでき、その後でこれらのオブジェクトを指定するのにこれらの名前を使用できる。名前付け機能と統合する部分である**位置特定機能 (locating facility)** はオブジェクトの名前を分散システムにおけるオブジェクトの位置に写像する。名前付け及び位置特定機能は併合して**名前付けシステム (naming system)** を形成する。名前付けシステムは、ネットワークにおいてオブジェクトが実際にどのように、及びどこに位置しているのかについての詳細を隠蔽したオブジェクトの1つの抽象をユーザに提供する。オブジェクトの複製を扱う場合には抽象のさらに進んだレベルを提供する。オブジェクトの名前が与えられると、オブジェクトの複製の一式の位置を戻す。

分散システムにおける**位置透過性 (location transparency)** の目標を達成する際に、名前付けシステムは非常に重要な役割を果たす。透過的移動及びオブジェクトの複製の実現に加えて、名前付けシステムはオブジェクト共有をも実現する。もし各種の計算を同じオブジェクトに対して行いたいならば、それぞれにオブジェクトの名前を含ませることにより実現できる。各計算に含まれる名前は必ずしも同じではないかもしれないが、この場合には同じオブジェクトに写像される。本章では、分散システムにおける名前付けシステムの設計及び実装についての各種の取り組みを述べる。

10.2 良好な名前付けシステムの望ましい特性

分散システムにおける良好な名前付けシステムは以下に述べる特性をもつべきである。

1. **位置透過性 (location transparency)**: **位置透過性**は、オブジェクトの名前がオブジェクトの物理的位置に関するどのようなヒントも持っていないことを意味する。すなわち、あるオブジェクトの名前は、システムの物理的接続又はトポロジーに、あるいはオブジェクトの現在位置に独立であるべきである。

2. **位置独立性 (location independency)**: 性能、信頼性、可用性、及びセキュリティのために、分散システムは、システムの様々なノードの間で、オブジェクトの移動及び再配置を可能とするオブジェクト移動の機能を提供する。**位置独立性**は、オブジェクトの位置が変化してもオブジェクトの名前は変更する必要がないことを意味する。さらには、ユーザは、ユーザがオブジェクトをどのノードからアクセスするかに関わりなく、同じ名前によってオブジェクトをアクセスできるべきである。それゆえ、位置独立性の要求は、次の2つの特性をもつ全体**名前付け機能 (global naming facility)** を必要とする。

■ どのノードにあるオブジェクトも、その物理的位置の知識がなくてもアクセスされることができる (要求を受けるオブジェクトの位置独立性)。

■ どのノードにあるオブジェクトも、自己の物理的位置の知識がなくてもアクセス要求を発行することができる (要求を発行するオブジェクトの位置独立性)。この特性は**ユーザ移動性 (user mobility)** としても知られている。

位置独立の名前付けシステムは、同じオブジェクト名前を2つの異なる時間の瞬間において異なる位置に写像できる**動的写像方式 (dynamic mapping scheme)** をサポートしなければならない。それゆえ、位置独立性は位置透過性よりも強い特性である [Levy 及び Silbershatz 1990]。

3. **規模拡張性 (scalability)**: 分散システムは数個のノードをもつシステムから多数のノードをもつシステムまでサイズの範囲が様々である。さらには、分散システムは通常オープンシステムであり、それらのサイズは動的に変化する。それゆえ、どれほどの大きさの名前の集合を取り扱うべきかについての事前の考えを確実にもっておくことは不可能である。したがって、通常は名前空

間のサイズの変更につながる分散システムの動的な規模の変化に、適応することができなければならない。すなわち、システム規模の変化によって、名前付け又は位置特定機構に何ら変更を要してはならない。

4. **統一名前付けコンベンション**:多くの現存システムにおいて、異なるオブジェクトの名前付け方法、**名前付けコンベンション (naming convention)** と呼ばれる、が異なるタイプのオブジェクトの名前付け用に使用されている。例えば、ファイル名前は典型的にユーザ名前やプロセス名前とは異なる。このような非統一名前付けコンベンションを使用する代わりに、良好な名前付けシステムはシステムのすべてのタイプのオブジェクトに同じ名前付けコンベンションを使用すべきである。

5. **同一オブジェクトへの複数ユーザ定義名前**: 1つの共有オブジェクトに対して、オブジェクトの異なる使用者がアクセスするために、彼ら自身の便利な名前を使用できることが望ましい。したがって、名前付けシステムは、同じオブジェクトに複数のユーザが定義した名前を割り付ける柔軟性を提供しなければならない。この場合には、あるユーザが他のユーザからの名前に影響を与えることなく、オブジェクトの名前を変更又は削除することが可能である。

6. **グループ名前付け**: 名前付けシステムは、多くの異なるオブジェクトを同じ名前で特定することもできるべきである。このような機能はブロードキャスト機能をサポートすること、及び協議や、その他のアプリケーションのためのグループオブジェクトを扱うために有用である。

7. **有意な名前**: 名前はあるオブジェクトを特定するどのような文字列であってもよい。しかし、ユーザにとって、メモリポインタ、ディスクブロック番号、又はネットワークアドレスのような低いレベルの識別子よりも意味のある名前が好まれる。これは、有意な名前は典型的には対象の内容又は機能に関するものを含み、ユーザ間で容易に転送でき、そして容易に記憶したり使用したりできるからである。それゆえ、良好な名前付けシステムは少なくとも 2 つのレベルの**オブジェクト識別子**、1つは人間のユーザに便利なものと、もう1つはマシンに便利なものをサポートすべきである。

8. **性能**: 名前付けシステムの最も重要な性能の測度は、オブジェクトの名前

をオブジェクト位置のようなその属性に変換するのに必要な時間である。分散環境においては、この性能は、名前写像演算の間にメッセージが交換される回数によってほぼ定まる。それゆえ、名前付けシステムは、名前写像演算の間にメッセージの交換される回数ができる限り少ないという意味で効率的であるべきである。

9. **フォールトトレラント性**: 名前付けシステムは、分散システムネットワークにおけるノード又は通信リンクの故障によって生じる障害に、ある範囲で、耐えることができるべきである。すなわち、名前付けシステムはこれらの故障の場合にも、恐らく縮退した形で、機能を継続すべきである。縮退は性能面、機能面、又はその両面で起きうるが、その原因となる故障に、ある意味では、対応しているべきである。

10. **複製透過性**: 分散システムにおいて、性能及び信頼性を改善するために一般に、あるオブジェクトの複製が生成される。名前付けシステムは、ユーザに透過的な方法で、同じオブジェクトの複数のコピーの使用をサポートすべきである。すなわち、必要がなければ、ユーザはあるオブジェクトの複数コピーを使用していることに気付くべきではない。

11. **最近傍複製の位置特定**: 名前付けシステムが同じオブジェクトの複数コピーの使用をサポートするとき、名前付けシステムにおける**オブジェクト位置特定 (object-locating) 機構**は所望のオブジェクトの最も近い複製の位置を常に供給すべきであることは重要である。これは、もしオブジェクト位置特定機構がこの点を考慮していないと、オブジェクトアクセス操作の効率が影響を受けるからである。これは図 10.1 に例が図示されている。この図では、所望のオブジェクトがノード N_2、N_3、及び N_4 に複製されていて、オブジェクト位置特定機構が、ノード N_2 に存在する最も近い複製の代わりにノード N_4 に存在する複製に写像する。明らかにこれは望ましくない。

12. **全複製の位置特定**: オブジェクトの最近傍複製に位置特定する代わりに、信頼性の観点から所望のオブジェクトのすべての複製がオブジェクト位置特定機構によって位置特定されることも重要である。この特性の必要性を図 10.2 に示す。この図では、ネットワークの通信リンクの故障により、ノード N_5 最近傍の複製が現在アクセスできない状態を示している。この場合に、離れたノー

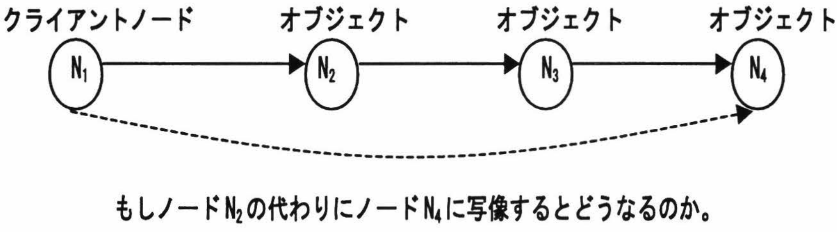


図 10.1　オブジェクトの最近傍複製の位置特定の重要性

ド N_4 に存在するもう 1 つのオブジェクトの複製を使用することができる。

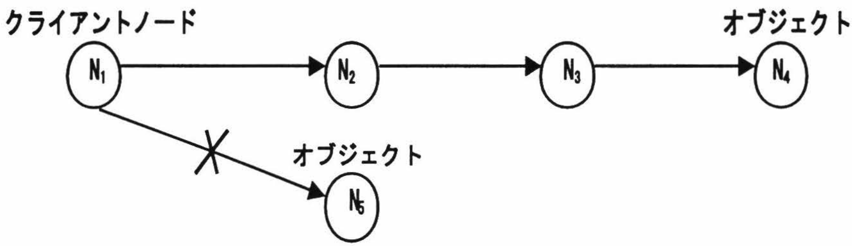


図 10.2　オブジェクトの全複製の位置特定の重要性

10.3　基本的な用語と概念

名前付けシステムは分散オペレーティングシステムの最も重要な要素の 1 つである。その理由は、統一的な方法で、サービスやオブジェクトを特定したり、アクセスしたりすることを名前付けシステムが可能にするからである。名前の重要性にも拘わらず、文献には、一般的な統一された扱いが見当たらない。本節では、分散システムにおけるオブジェクト名前付けに関連する基本的な用語と概念についての定義と説明を行う。

10.3.1　名前

名前 (name) は有限のアルファベットから選ばれた一式のシンボルからなる文字列である。例えば、SINHA、#173#4879#5965、node-1!node-2!node-3!sinha、/a/b/c、25A2368DM197 などはすべて ASCII 文字セットからのシン

ボルからなる正当な名前である。名前は、オブジェクトを指定したり、特定したりするために使用されるので、**識別子 (identifier)** とも呼ばれる。名前はまた、物理的オブジェクトの集まりの間から1つの物理的オブジェクトを特定する論理的オブジェクトとしてもみなすこともできる。それゆえ、名前とオブジェクトとの対応は、オブジェクト特定の目的で、論理的及び物理的オブジェクトを結び付ける関係である。

10.3.2 人間指向及びシステム指向名前

名前は、システムアーキテクチャのすべてのレベルにおけるオブジェクトを指定したり又は参照したりするために使用される。名前は、それらが定義されるレベルに応じて様々な目的、形式、及び特性を持っている。しかし、オペレーティングシステムで広く使用されている2つの基本的クラスに非公式に区別される。それらは、**人間指向 (human-oriented) 名前 システム指向 (system-oriented) 名前**ある。

人間指向名前は、一般にそのユーザにとって意味のある文字列である。例えば、/user/sinha/project-1/file-1は人間指向名前である。人間指向名前はそれらのユーザによって定義される。共有されるオブジェクトに対して、そのオブジェクトの異なるユーザがそのオブジェクトをアクセスするためにそのオブジェクトに対して彼ら自身の人間指向名前を定義する柔軟性をもたなければならない。一人のユーザがオブジェクトに対する各自の名前を、他人の名前に影響を与えることなく、変更又は削除したりできる柔軟性もまた必要である。透過性のために、人間指向名前は指定するオブジェクトの物理的な位置又は構造に対し独立しているべきである。人間指向名前は、それらはユーザによって容易に記憶されるので、**高レベル (high-level name)** 名前として知られている。

人間指向名前はオブジェクトに対してユニークではないし、異なるオブジェクトに対してだけではなく、同じオブジェクトに対する異なる名前に対しても、通常長さが変化する。したがって、名前を特定するための目的でマシンによって操作したり、格納したり、使用したりすることは容易ではない。さらには、あるレベルで全体システムにおけるそれぞれのオブジェクトはユニークに特定されなければならない。それゆえ、ユーザにとって有用である人間指向名前に加えて、システムによって効率的に使用されるためのシステム指向名前が必要である。これらの名前は一般に固定長のビットパターンであり、マシンによって容易に操作されたり、格納されたりする。これらはシステムによって自動的

に生成される。これらは、集中型の識別子生成者の効率及び信頼性の問題を避けるために、分散型の方法で生成されるべきである。システム指向名前は基本的にはシステムによって使用されることを意味しているが、ユーザによって使用されることもあり得る。これらは**ユニーク識別子 (unique identifier)** 及び**低レベル名前 (low-level name)** として知られている。

図 10.3 はこれらの 2 つの名前のタイプに基づく単純な名前付けモデルを示す。この名前付けモデルにおいて、人間指向名前は最初にシステム指向名前に写像 (変換) され、それから対応するオブジェクトの複製の物理的位置に写像される。

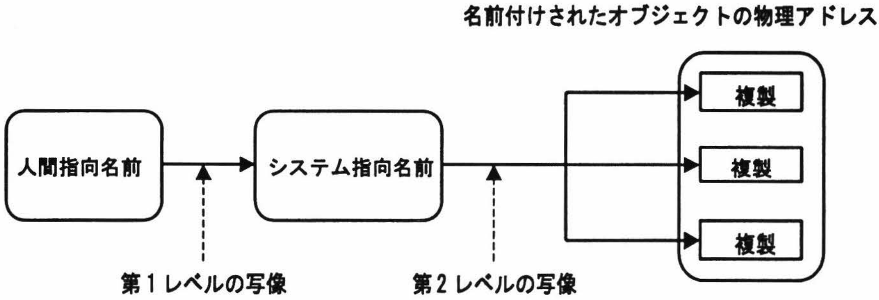


図 10.3　分散システムにおいて人間指向及びシステム指向名前の使用に基づく単純名前付けモデル

10.3.3　名前空間

名前付けシステムはオブジェクトに名前を割り付けるために 1 つ又はそれより多くの名前付けコンベンションを採用する。例えば、名前付けシステムは、オブジェクトに人間指向名前を割り付ける 1 つの名前付けコンベンション、及びオブジェクトにシステム指向名前を割り付けるもう 1 つの名前付けコンベンションを使用するかもしれない。名前の意味論的解釈 (semantic interpretation) と同様に構文的表現 (syntactic representation) はその名前に使用されている名前付けコンベンションに依存する。与えられた名前付けコンベンション対応する名前の集合は**名前空間 (name space)** を形成すると言われる [Terry 1984]。

10.3.4 フラット名前空間

最も単純な名前空間は、名前が構造を持たない文字列である**フラット名前空間 (flat name space)** ある。フラット名前空間において定義された名前は**プリミティブ (primitive)** 又は**フラット名前 (flat name)** 呼ばれる。フラット名前は構造を持たないので、明確で意味のある名前を大規模のオブジェクト集合に割り付けることは困難である。それゆえ、フラット名前はわずかな数のオブジェクトの名前をもつ小規模の名前空間、又はユーザにとって意味をもつ必要のないシステム指向名前に使用するのに適している。

10.3.5 分割名前空間

オブジェクトの大規模な集合に明確で意味のある名前を割り付ける必要があるときは、名前空間を分離されたクラスに分割する名前付けコンベンションが通常使用される。分割が統語論的 (syntactically) になされるならば、通常はそうであるが、名前構造は物理的又は組織上の連想を反映する。**分割名前空間 (partitioned name space)** の各部分は名前空間のドメイン **(domain)** 呼ばれる。

分割名前空間の各ドメインはそれ自体でフラット名前前空間と見られ、ドメイン内で定義された名前はそのドメイン内でユニークであるかもしれない。しかし、2 つの異なるドメインは、それらの中で定義された共通の名前を持つかもしれない。ドメインの中で定義された名前は**単純名前 (simple name)** と呼ばれ、**複合名前 (compound name)** は全体でユニークな特定のために使用される。複合名前は 1 つ又はそれ以上の単純名前からなっている。単純名前間は、/、$、@、%他のような特別の**区切り文字 (delimiter character)**(本章の例や説明では UNIX のファイルシステムコンベンションにしたがって、区切り文字/が使用される) によって分けられている。例えば、/a/b/c は 3 つの単純名前 a、b、及び c からなる複合名前である。

分割名前空間の通常よく使用されるタイプは**階層型名前空間 (hierarchical name space)** である。ここでは名前空間は複数レベルに分割され、逆転ツリー状 (inverted tree) に構造化される。名前空間ツリーの各ノード (グラフ上の) は名前空間のドメインに対応する。このタイプの名前空間において、レベルの数は固定のこともあるし、可変のこともある。例えば、Grapevine システムは名前空間ツリーを 2 レベルで管理する [Birrell 他 1982] が、Xerox の

Clearinghouse においては、名前空間ツリーは 3 レベルである [Oppen 及び Dalal]。他方、DARPA のインターネットドメイン名前付けシステム (Internet Domain Naming System)[Su 及び Postel] 及び [Lantz 他 1985] に提案されているユニバーサルディレクトリサービス (Universal Directory Service) において、名前空間ツリーは任意の数のレベルをもつことができる。階層型名前空間において定義された名前は**階層型名前 (hierarchical name)** と呼ばれる。

階層型名前は長年ファイルシステムにおいて使用されてきたが、近年、分散システムにおける他のオブジェクトを名前付けするためにも採用されてきた。階層型名前のいくつかの例は我々の日常生活においても散見される。例えば、国及び地域コードを含む全体に拡張された電話番号は 4 レベルの階層型名前空間を形成する。コンピュータネットワークにおけるネットワークアドレスは、ネットワーク番号、ノード番号、及びソケット番号の 3 レベルの階層型名前空間を形成する。

10.3.6　名前サーバ

名前空間は名前サーバによって管理される。**名前サーバ (name server)** は名前付けされたオブジェクトについての情報を維持し、その情報をユーザがアクセスできるような機能を提供するプロセスである。これはオブジェクトの名前とオブジェクトの位置を含む特性とを結び付ける [Terry 1984]。

実際には、分散システムにおいて、オブジェクト名前の名前空間を取り扱うために、複数の名前サーバが通常使用される。各名前サーバは通常、分散システムにおけるオブジェクト集合の小規模なサブセットのみについての情報を持っている。あるオブジェクトについての情報を格納している名前サーバは、そのオブジェクトの**権限名前サーバ (authoritative name server)** とよばれる [Terry 1984]。各々の名前付けされたオブジェクトに対する権限名前サーバを決定するために、各オブジェクトの権限名前サーバのリストを含む**権限属性 (authority attribute)** を名前サービスは維持する。

分割名前空間は、フラット名前空間に比べて効率的に管理することが容易である。その理由は、各名前サーバに必要な構成データは、各個別のオブジェクトに対してではなく、各ドメインに対してであるので、構成データの量を減少することができるからである。例えば、階層型名前空間において、各名前サーバは、名前ツリーのルートドメインに対する権限名前サーバを位置特定する情報のみがあれば十分である。それに対応して、ルートドメインの権限名前サー

バは、ルートドメインから枝別れするドメインの権限名前サーバの位置を知っているべきである。一般に、あるドメインの権限名前サーバは、そのドメインから枝別れするドメインのみについての権限名前サーバの位置を知っているべきである。例えば、図 10.4 の名前空間ツリーにおいて、すべての名前サーバはドメイン D_1 の権限名前サーバの位置を知らなければならない。ドメイン D_1 の権限名前サーバはドメイン D_2、D_3、及び D_4 の権限名前サーバの位置を知っているのみでよい。そして、ドメイン D_2 の権限名前サーバはドメイン D_5 及び D_6 の権限名前サーバの位置を知っているのみでよい。それゆえ、階層の各種のレベルの名前サーバで維持されるべき構成データは、名前空間ツリーの枝別れの程度に比例する。この理由で、幾つかのレベルをもつ階層型名前付けコンベンションは多数のオブジェクトを名前付けするのに適している。

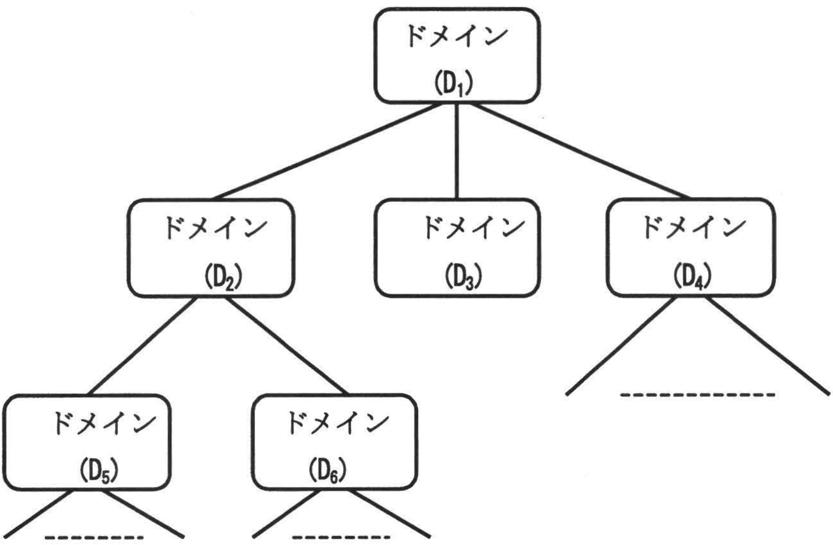


図 10.4 階層型名前空間のドメイン

重要なことであるが、名前サーバの内部表現は抽象的名前付けシステムの構造を反映することは必ずしも必要ではないことに注意されたい。例えば、階層型内部構造を採用する代わりに、名前サーバは階層型名前をフラットデータ構造に圧縮するかもしれないし、一連の単純名前要素の列の一致を調べる代わりに全体の複合名前の文字列で比較するかもしれない。実際に、幾つかの名前サーバは、名前空間の与えられたドメイン内で結び付け **(binding)** を実装する

のために、テーブル準拠の方法の代わりに手続き準拠の方法を使用している。しかし、名前サーバとドメインとの 1 対 1 対応は、分割名前空間の管理を容易にする。

10.3.7　名前エージェント

名前サービスの分散及び名前サーバの位置特定は、名前サービスのクライアントにとって透過的であるべきである。この透過性は、名前サーバ及びそれらのクライアントとの間で動作する**名前エージェント (name agent)** によって達成される。名前エージェントは存在する名前サーバの知識を維持する。あるクライアントがある名前サービスを要求すると、名前エージェントは、ユーザの要求を適切な名前サーバに転送するために、適切な通信プロトコルを使用する。サーバからの応答を受信すると、それをクライアントに転送する。

名前エージェントには 2 つのタイプがある。(a)1 つのクライアントのために働くプライベートタイプ、そして (b) 幾つかのクライアントのために働く共有タイプの 2 つである。プライベート名前エージェントは、クライアントプログラムにリンクされる 1 式のサブルーチンとして構成される。他方、共有名前エージェントは、名前サービス操作を起動するシステム呼び出しを伴いオペレーティングシステムのカーネルの一部として構成されるか、又はプロセス間通信プリミティブ経由でアクセスされる分離したプロセスとして構成される。

名前エージェントはいろいろな名前で知られている。例えば、CSNET の Name Server では、「名前サーバエージェントプログラム」と呼ばれ、DARPA の Internet Domain Name System(DNS) では、「**リゾルバ (resolver)**」と呼ばれ、DCE の Directory Service では、「**クラーク (clerk)**」と呼ばれ、そして COSIE の Name Server では「ユーザインタフェース」と呼ばれる。

10.3.8　コンテキスト

名前は常に**コンテキスト (文脈)** を伴う。コンテキスト (context) は名前が正当 (valid) である環境と考えることができる。すべての名前はあるコンテキストに対して相対的に解釈されるので、context/name の対は**確認名前 (qualified name)** と言われ、あるオブジェクトをユニークに特定するのに使用される。

コンテキストの概念は名前空間をより小さい部分に分割するのに非常に有用であることが証明されている。しばしば、コンテキストは、自然地理的、組織

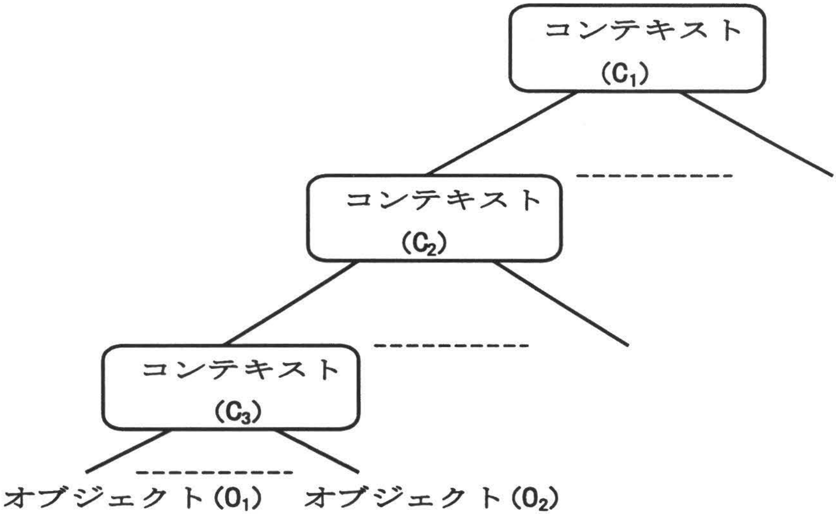


図 10.5　入れ子のコンテキスト

的、又は機能的境界に沿った名前空間の分割を表す [Terry 1986]。分割名前空間において、各ドメインは名前空間のコンテキストに対応する。あるコンテキストにおける名前は、他のコンテキストにどんな名前が存在するかに関わりなく生成されることができる。それゆえ、ある名前は 1 つ以上のコンテキストに出現するかもしれない。階層型名前空間におけるように、コンテキストは入れ子 (nest) になっているかもしれない。例えば、図 10.5 の名前空間ツリーにおいて、コンテキスト C_3 はコンテキスト C_2 の中に入れ子になっており、C_2 はコンテキスト C_1 の中に入れ子になっている。入れ子になったコンテキストにおいて、確認名前は一連の名前要素からなっている。各名前要素はそれぞれコンテキスト、サブコンテキスト、サブサブコンテキスト、… 、そして最後のコンテキストにオブジェクトの名前を指定している。例えば、図 10.5 において、コンテキスト C_3 に関連するオブジェクト O_1 の確認名前は、$C_1/C_2/C_3/O_1$ となる。

名前管理のために、名前付け情報データベースを複数の名前サーバに分散できるように、コンテキストは名前付け情報データベースを分割する手段を提供する。コンテキストは、名前付けされたオブジェクトに関する情報の格納及び複製のための不可視の単位を表す [Terry 1986]。

10.3.9　名前リゾルーション

名前リゾルーション (name resolution) はあるオブジェクトの名前をオブジェクトの位置などの属性 (property) に写像する処理である。オブジェクトの属性はそのオブジェクトの権限名前サーバによって格納及び維持されるので、名前リゾルーションは基本的にはオブジェクト名前をそのオブジェクトの権限名前サーバに写像する処理である。オブジェクトの権限名前サーバが位置特定されて、次に、オブジェクトの属性を読み取ったり、更新したりするための処理が起動される。

分散システムにおける各名前エージェントは、事前に少なくとも 1 つの名前サーバの存在を知っている。名前を変換するために、クライアントは最初にその名前エージェントにコンタクトする。次に名前エージェントは既知の名前サーバにコンタクトし、その名前サーバは他の名前サーバにコンタクトする。名前サーバは互いに特定できなければならないので、それらもまたオブジェクトとして取り扱われ、名前が付けられている。しかし、巡回を避けるため、名前付けシステムの与えられたレベルでの名前サーバは一般に、システム指向の低レベル名前で互いを特定する。

分割名前空間において、名前リゾルーション機構は、名前付けされたオブジェクトの権限名前サーバに遭遇するまで、1 つのコンテキストから別のコンテキストへと、名前リゾルーションのチェーンをたどっていく。図 10.6 に示されているように、与えられた名前は、それが関連するコンテキストにおいて最初に解釈 (interpret) される。この解釈は、名前付けされたオブジェクトの権限名前サーバを戻すか、又はその名前を解釈するための新しい名前と新しいコンテキストを戻すかのいずれかである。前者の場合、名前リゾルーション処理は完了し、後者の場合、解釈処理は、名前付けされたオブジェクトの権限名前サーバに遭遇するまで継続する。もし、名前の変換に含まれるすべてのコンテキストが 1 つの名前サーバによって管理されるなら、完全な名前リゾルーション操作は 1 つの名前サーバによって実行され、そうでばければ、複数の名前サーバは名前リゾルーション操作に関ってくる。

10.3.10　略称／別名

分割名前空間の場合に、確認名前 (通常は複合名前) は幾つかの単純名前からなり、非常に長い名前であるかもしれない。ユーザにとってオブジェクトを使

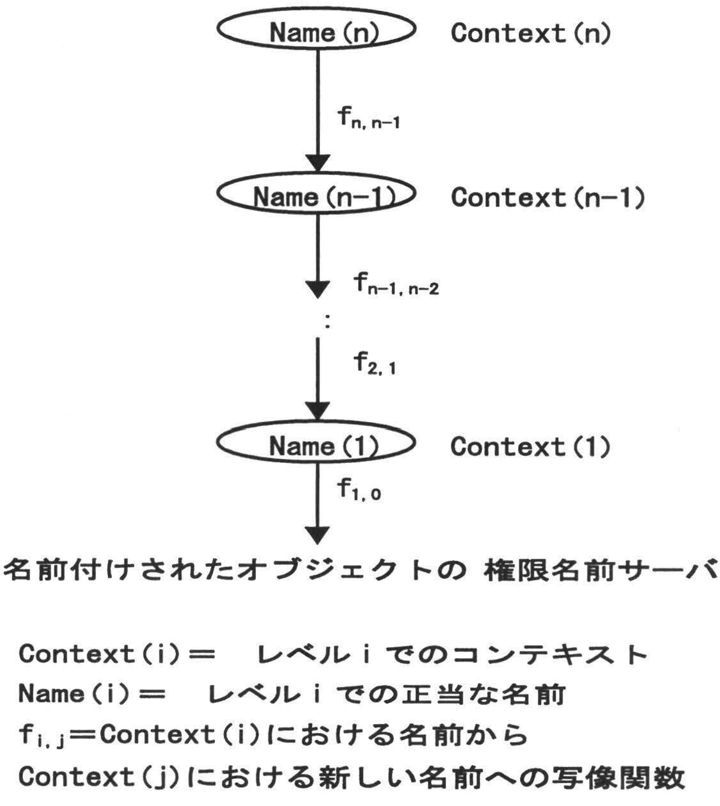


図 10.6 コンテキストごとの名前リゾルーション機構

用するたびに毎回、確認名前を指定することは不便であるかもしれない。それゆえ、分割名前空間に対する名前付けコンベンションにおいて通常、確認名前の短縮形の代用品をユーザ自身で指定し使用することが許されている。これらの短縮形は**略称 (abbreviation)** と呼ばれる。ユーザによって定義されたすべての略称はそのユーザ個別のコンテキストを形成する。略称の確認名前への写像は個別のコンテキストベースで維持される。あるユーザが略称を指定するとき、そのユーザの個別コンテキストに関連する写像を用いて、確認名前に変換される。それゆえ、異なるユーザは異なるオブジェクトに同じ略称を使用することもある。例えば、2 人のユーザ、**ユーザ 1(user-1)** 及び**ユーザ 2(user-2)** が、それぞれ確認名前**/user-1/project-1/group-1/object-1** 及び**/user-2/project-5/group-3/object-1** をもつオブジェクトを特定するために、同じ略称 **my-object-1** を使用するかもしれない。略称は**別名 (alias)** としても知られている。別名は一般的に、1 つの個別コンテキストの中においてのみユニークであることが必要な単純名前であることに注意されたい。UNIX ファイルシステムのシンボリックリンク機能は名前付けシステムにおける別名付け機能の一例である。

別名付けの場合に、結び付けは 1 つの単純名前を確認名前に関連付ける。しかし、ユーザは 1 つのオブジェクトを、2 つ又はそれ以上の単純名前の 1 つによって特定しようと望むかもしれない。ある名前付けシステムでは、コンテキスト内において多対 1(many-to-one) の結び付けを認めることにより、この柔軟性を提供するかもしれない。すなわち、1 つ以上の単純名前が、与えられたコンテキスト内で、同じ確認名前に結び付けられるかもしれない。もしこの機能が提供されると、同一の確認名前に結び付けられる単純名前は**同義 (同名、シノニム)(synonym)** 又は**ニックネーム (nickname)** と呼ばれる。

10.3.11　絶対名前及び相対名前

ツリー構造名前空間におけるオブジェクトの**完全確認名前 (full qualified name)** を指定するのを避ける別の 1 つの方法は、**現在動作中コンテキスト** (current working context) の概念の使用である。現在動作中コンテキストはより短い、**現在コンテキスト (current context)** 又は**動作中コンテキスト (working context)** としても知られている。この概念によると、あるユーザは常に現在コンテキストと関連付けられている。あるユーザは自分の望むようにいつでも現在コンテキストを変更できる。

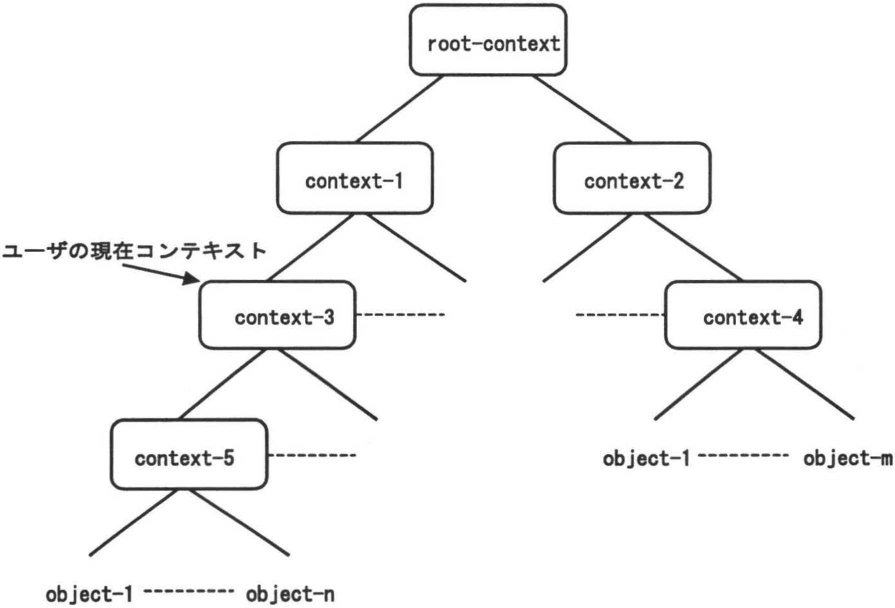


図 10.7　絶対名前及び相対名前を説明するためのツリー構造名前空間

この方法において、オブジェクト名前は絶対と相対の 2 つのタイプをとりうる。**絶対名前 (absolute name)** は名前空間ツリーのルートコンテキストに始まり、指定されたオブジェクトまで、パス上のコンテキスト名前を指定して、パスを下がっていく。他方、**相対名前 (relative name)** は現在のコンテキストから指定されたオブジェクトへのパスを定義する。これは、ユーザの現在のコンテキストから始まり「相対的」であるので、相対名前と呼ばれる。例えば、図 10.7 のツリー構造の名前空間において、もしユーザの現在コンテキストが、**root-context/context-1/context-3** であるならば、相対名前 **context-5/object-1** は絶対名前 **root-context/context-1/context-3/context-5/object-1** と同じオブジェクトを参照 (指定) する。この方法では、あるコンテキストの親コンテキストを参照する手段も提供される。例えば、ツリー構造名前空間を使用する UNIX ファイル名のコンベンションにおいて、2 つのピリオド (.. 、「ドットドット」と発音される) がディレクトリの親ディレクトリ (ディレクトリは基本的にコンテキストである) を示す。ルートディレクトリはそれ自身の親である。

この方法において、ユーザは次の方法のいずれかでオブジェクトを指定できる。

1. 完全 (絶対) 名前を使用する
2. 相対名前を使用する
3. 現在のコンテキストをまず変更し、それから相対名前を使用する

例えば、図 10.7 において、もし現在コンテキストが **root-context/context-1/context-3** であるユーザが、**root-context/context-2/context-4/object-1** のオブジェクトを指定しようとするとき、ユーザは以下の方法により指定する。

1. そのオブジェクトの絶対名前、すなわち **root-context/context-2/context-4/object-1** を指定する。

2. 現在コンテキストに相対的なオブジェクト名前として、**../../context-2/context-4/object-1** を指定する。この相対名前において、最初のドットドットは **context-1** の親コンテキストを示し、第 2 のドットドットは **context-1** の親コンテキスト、すなわちルートコンテキスト **root-context** を示す。

3. 上記の 2 つの名前指定方法において、相対名前が、ある場合には、絶対名前と同じ長さ、又はより長い名前になることもあり得る。これでは相対名前を使用する意味がなくなる。そこで、第 3 の方法は、ユーザの現在コンテキストを最初に変更し、それから新しいコンテキストに相対的な名前によってオブジェクトを参照する。例えば、上記の例では、ユーザは最初にコンテキスト変更コマンドを使用して、**root-context/context-2/context-4** という名前のコンテキストに変更し、それからユーザの現在のコンテキストに相対的な単純名前 **object-1** によって単純にオブジェクトを指定する。

10.3.12　総称及びマルチキャスト名前

シノニムの使用が多対 1 の結び付きをサポートする名前付けシステムを必要とすることは既に述べた。他方、総称及びマルチキャスト名前は **1 対多 (one-to-many)** の結び付きをサポートする名前付けシステムを必要とする。

すなわち、名前付けシステムは１式の確認名前に対して１つの単純名前を結び付けることができなければならない。

総称名前付け (generic naming) 機能において、結び付けられているオブジェクトの集合のどれか１つに、名前が写像される。このタイプの機能は、要求をサービスできる幾つかのサーバのどれかにユーザがサービスを要求する場合で、ユーザがどのサーバで要求がサービスされるかに関心がない場合に有用である。

グループ又はマルチキャスト名前付け (group 又は multicast naming) 機能において、結び付けられているオブジェクトの集合の要素のすべてに、名前が写像される。このタイプの機能はメッセージのブロードキャスト及びマルチキャストに特に有用である。

10.3.13 記述的／属性準拠名前

記述的／属性準拠名前 (descriptive/attributed-based name) をサポートする名前付けシステムでは、オブジェクトを記述し、名前空間におけるオブジェクトの集合からユニークにオブジェクトを特定する１式の属性 (attribute) 又は固有特性 (property) によって、オブジェクトを特定することができる。固定の構造を (すなわち固定の１式の属性) もっている分割名前空間は記述的名前付けコンベンションの一例である。この場合には、名前空間の各ドメインはオブジェクトの属性を定義するために使用される。１つの属性は１つのタイプ (type) と１つの値 (value) の両方をもっている。タイプは値フィールドのフォーマットと意味を表す。例えば、ファイルオブジェクトをユニークに特定するのに使用される典型的な１式の属性は、**User = Sinha, Creation date = 1995/04/06, File type = Source, Language=Pascal, Name=Quicksort** である。

１つの属性の値は幾つかのオブジェクトに対して同じであるかもしれないが、同時に考慮される属性の組み合わせによって、１つのオブジェクトを参照することに注意されたい。さらには、あるオブジェクトを特定するのに、名前付けコンベンションのすべての属性を指定する必要は必ずしもない。属性準拠名前付けシステムは通常、目標オブジェクトがユニークに特定されるのに十分な属性が、参照時に指定されなければならないという考えに基づき動作する。記述的名前付けコンベンションを使用する分割名前空間において、ドメインはどのような形式でも設定できる。

マルチキャスト又はグループ名前付け機能は、属性準拠名前付けにおいて、名

前のリストに対応する属性を構築することにより、容易に提供される。グループ名前はメール配信リスト及びアクセス制御リストの作成に特に有用である。

10.3.14　送信元ルーティング名前

多くの名前空間は使用している物理ネットワークの構造を反映している。名前空間の構造が、分散システムで使用しているネットワークと同じ形式をもっているとき、名前空間は**送信元ルーティング名前 (source-routing name)** を定義する。送信元ルーティング名前は分散システムのネットワークを通るパスを特定する。UNIX-to-UNIX・Copy(UUCP) 名前空間は、例えば **host-1!host-2!host-3!sinha** のような送信元ルーティング名前空間形式の名前を定義する。送信元 (source) コンピュータによってネットワークを通るルートが指定されるので、UUCP スタイル名前は送信元ルーティング名前と呼ばれる。例えば、上記の例で、指定されたルートは **host-1** から、**host-2**、**host-3**、**sinha** となる。UUCP スタイル名前は、**開始点 (starting point)** に対して相対的に解釈されねばならないので、相対名前である。

10.4　システム指向名前

システム指向名前 (system-oriented name) は次の特徴をもつ。

1. システム指向名前は大きい整数又はビット列である。例えば、典型的な名前付けシステムにおいて、システム指向名前は、$2^{128}-1$ のような大きい整数集合の要素であるかもしれない。

2. ほとんどの名前付けシステムにおいて、空間及び時間の両方でユニークであることが保証されているので、システム指向名前は**固有識別子 (unique identifier)** とも言われる。すなわち、これらの名前は生存期間 (lifetime) の間変化せず、そして一度使用されると、(別のオブジェクトに) 再使用されることはない。それゆえ、上記の名前付けシステムにおいて、128 ビットパターンはどれにも対応していないか、又は対応しているならば、いつも同じものに対応する。これが固有識別子が長いビット列となっている理由である。

3. 長さが可変である人間指向名前とは異なり、ある名前付けシステムのすべ

てのシステム指向名前は、これらの名前によって特定されるオブジェクトのタイプや位置によらず同じ長さである。これがすべてのオブジェクトの名前付けを統一的にしている。

4. ある名前付けシステムのすべてのシステム指向名前は統一的な長さで、通常は人間指向名前よりも短いので、名前へのハッシュ、ソートなどの操作が容易に行える。そこで、システム指向名前はマシンによって効率的に処理するのに適している。

5. システム指向名前は推察が困難であり、それゆえ時々セキュリティ関係の目的にも使用される。例えば、これらはオブジェクトをアクセスするための許可 (permission) の切符として使用される。名前を正当と判断する計算は、推察や考案されたものというよりも、故意にあたえられたものと仮定する。これが保護された名前 (protected name) の使用、又はアクセス制御管理の古典的な方法の一つであるケーパビリティ **(capability)** の一例である。

6. システム指向名前はシステムによって自動的に生成される。

システム指向名前に関係する重要な問題は、システムにおいて各オブジェクトにどのようにしてユニークな名前を生成するかという点である。この問題を扱う方法を述べる前に、システム指向名前は、非構造化か構造化かのどちらである (図 10.8 に各タイプの例を示す)。非構造化名前は、あるオブジェクトをユニークに特定するのみで、そのオブジェクトについての他の情報を提供しない大きい整数又はビット列の単一のフィールドをもっている。他方、構造化名前は 1 つより多い要素を含んでいる。その幾つかは、その名前によって特定されるオブジェクトの特性についての情報を提供する。例えば、構造化名前はそのオブジェクトが生成されたノード、又はオブジェクトが現在存在しているノードを指定する要素をもっているかもしれない。これらの 2 つのタイプのシステム指向名前に対する 2 つの基本的方法を以下に述べる。

10.4.1 集中型システム指向名前の生成

集中型の方法は構造化システム指向名前と同様に非構造化名前の両方の生成に使用され得るが、主に非構造化名前を使用する名前付けシステムに使用され

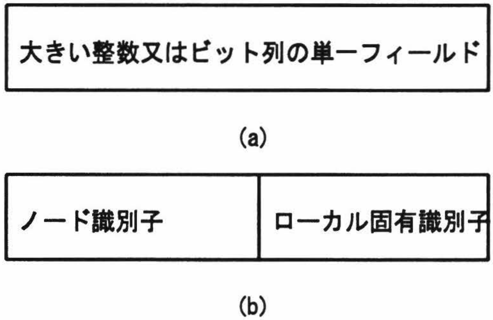


図 10.8　(a) 非構造化システム指向名前、(b) 構造化システム指向名前

る。この方法では、標準的で、統一的なグローバル識別子が、集中型グローバル固有識別子生成者によってシステムの各オブジェクトに対して生成され、このグローバル名前空間を各ノードのローカルドメインに分割するための何等かの方法が使用される。各ノードは、これらのグローバル識別子を各ノードで生成されたオブジェクトに直接的に結び付けるか、又は各ノードで生成されたオブジェクトのローカル識別子をこれらのグローバル識別子に写像するかのいずれかである。後者の場合、ローカル識別子のグローバル識別子への結び付けは永続的であるか又は一時的である。集中型の方法は、実装が単純で容易であることと、非構造化グローバル固有識別子を生成するただ 1 つの方法であるという利点をもっている。しかし、効率と信頼性の欠点をもっている。単一のグローバル固有識別子生成者は大規模名前空間の隘路となるかもしれない。また、グローバル固有識別子の連続的な生成は、集中型グローバル固有識別子生成者の信頼性に全面的に依存する。

10.4.2　分散型システム指向名前の生成

集中型の方法の効率の悪さと信頼性の低さといった欠点の対策として、システム指向名前の生成のための分散型の方法が幾つかの名前付けシステムで使用されている。しかし、分散型の生成によって、必然的に名前付けシステムは構造化オブジェクト識別子を使用することとなる。分散型の方法において、**階層的連接 (hierarchical concatenation)** の方法がグローバル固有識別子の生成のために使用される。この方法において、それぞれの**識別ドメイン (identification domain)** は固有識別子を割り当てられ、あるオブジェクトに

対するグローバル固有識別子は、ドメインの固有識別子とこのドメイン内で使用される識別子を連接することにより生成される。それゆえ、この方法で生成されたグローバル固有識別子は基本的に2つの要素からなっている。すなわち、ドメインの識別子とこのドメイン内での識別子である。それぞれの要素は2つ又はそれより多くのサブ要素からなっていることに注意されたい。例えば、広域ネットワーク (WAN) に対するドメイン識別子は2つのサブ要素からなっていることがある。すなわち、ネットワーク識別子とこのネットワーク内のノード識別子である。

次に生じる疑問は、ドメイン内の固有識別子をどのように効率的に発生させるかということである。これを行う1つの方法はシステムの各ノードを名前空間の1つのドメインとして扱い、ノードの実時間クロックの読み取り (タイムスタンプと呼ばれる) をノード内の識別子と扱う方法である。この方法によると、グローバル固有識別子は **(node-ID, time-stamp)((ノード ID、タイムスタンプ))** の対の形式をとる。別の方法は、各サーバをドメインとして扱い、サーバ特有のやり方で、処理するオブジェクトのためのオブジェクト識別子をサーバに生成させる方法である。この場合に、グローバル固有識別子は **(server-ID, server-specific-unique-identifier)((サーバ ID、サーバ特有固有識別子))** の形式をとる。

分散型の方法は集中型の方法に比べてよい効率と高い信頼性をもっているが、次の欠点がある。

1. 異種環境では、識別子の形式と長さは異なるコンピュータ (ノード) によって異なり、その結果、不統一のグローバル固有識別子となる。アプリケーションがこれらの低レベル識別子の不統一性を扱う準備をすることは弱点 (awkward) 又は非効率となりかねない。

2. この方式では、ノードの境界又はサーバが明示的に可視的となる。

10.4.3 クラッシュ発生時の固有識別子の生成

固有識別子 (グローバル又はローカルによらず) の生成に関係する別の重要な問題はクラッシュ発生時における固有識別子の生成が可能であるかという点である。クラッシュでは固有識別子生成者の状態情報が失われる。それゆえ、回復時に固有識別子生成者は正しく動作しないかもしれない。これがユニークで

ない識別子の発生という結果になり得る。この問題を解決するために、2 つの基本的方法が使用される [Watson 1981]。

1. **故障の前後で継続して動作するクロックの使用**. この方法では、固有識別子生成者のノードのクロックが使用される。このクロックはノードが故障しても動作を継続することが保証されており、必要な識別子がユニークでなければならない期間の間で巡回しないことになっている。しかし、この方法を実装するために、クロック間隔に必要な**粒度** (**granularity**) に依存して、長い識別子になることもある。

2. **2 レベル以上の格納装置の使用**. この方法では、2 レベル又はそれより多いレベルの格納装置が使用され、固有識別子は各レベルに 1 つのフィールドをもつ階層的な形で構造化される。各レベルに関連するカウンタは対応するフィールドに最大の値を含んでいる。これらのフィールドの現在値は主メモリにキャッシュされている。低レベルフィールドが巡回しようとするとき、又は関連する格納装置が故障するとき、次に高レベルのカウンタが増加され、低レベルのカウンタがリセットされる。もし安定格納装置 (故障でもその中の情報が生き残る) が使用されると、2 レベルの格納 (上位レベルが主メモリへの格納で、低レベルが安定格納装置への格納) が、安全で効率的な固有識別子生成者を形成するのに十分である。最初の機構に比べ、この機構は短い識別子を生成することができるが、実装は複雑となる。

10.5　オブジェクト位置特定機構

オブジェクト位置特定 (object locating) は、あるオブジェクトのシステム指向固有識別子 (略して UID) をオブジェクトの複製の位置に写像する処理である。**オブジェクト位置特定操作 (object-locating operation)** はオブジェクトアクセス操作 (**object-accessing operation**) とは異なり独立していることに気付かれるであろう。分散システムにおいて、オブジェクト位置特定は単にオブジェクトの位置、すなわち、オブジェクトが存在するノードを知るための処理である。他方、オブジェクトアクセスは、そのオブジェクトへの所望の操作 (例えば、読み取り、書き込み) を実行する処理を含んでいる。それゆえ、オブジェクトアクセス操作は、オブジェクト位置特定操作が正常に実行された後に開始する。

いくつかのオブジェクト位置特定操作が提案され、多くの分散オペレーティングシステムによって使用されている。これらの機構を以下に概説する。ある分散システムに対し、特定の機構が適しているか否かは、多くの要因によっている。それらの要因は、システムの規模、名前付けシステムによって使用されているUIDのタイプ、システムがオブジェクトの移動を認めているか否か、システムがオブジェクトの位置透過性をサポートしているか否か、等である。

10.5.1 ブロードキャストによる方法

この方法では、クライアントノードからあるオブジェクトに対しての要求をブロードキャストすることによって、オブジェクトは位置特定される。要求はすべてのノードによって処理され、現在そのオブジェクトを持っているすべてのノードがクライアントノードに応答する。Amoeba [Mullender 及び Tanenbaum 1986] は遠隔ポートを位置特定するのに、この方法を使用している。

この方法は単純であり、目標オブジェクトのすべての複製の位置が得られるので信頼性が高い。しかし、低い効率と低い規模拡張性の欠点がある。その理由は、各要求のために生成されるネットワークトラフィックの量はシステムのノードの数に直接的に比例し、大規模ネットワークにとって致命的ともなる。それゆえ、この方法はノードの数が少ない場合で、通信速度が高く、かつオブジェクト位置特定要求頻度がそれほど高くない場合にのみ適している。

10.5.2 リングブロードキャスト拡大による方法

上記の純粋のブロードキャスト法は大規模ネットワークには向いていない。さらには、すべてのノードへの直接ブロードキャストは広域ネットワークによってサポートされていないかもしれない。それゆえ、**リングブロードキャスト拡大法 (expanding ring broadcast、ERB)**[Wiebe 1986] と呼ばれるブロードキャスト法の改良形が、ゲートウェイによって接続された構内ネットワーク (LAN) からなるインターネットワークにおいて、通常採用される。この方法では、オブジェクトが見つかるまで、又はすべてのLANの探索が終わるまで、距離の増加方向にLANが系統的に探索される。距離の測度はホップ **(hop)** が使用される。1つのホップはプロセッサ間の1つのゲートウェイに相当する。例えば、もしプロセッサAからプロセッサBへのメッセージが少なくとも2つのゲートウェイを通過しなければならないならば、AとBは2ホップ離れて

いる。同じ LAN の上のプロセッサはゼロホップ離れている。**リング (ring)** はあるプロセッサからある距離だけ離れた LAN の集合である。このようにして、$Ring_0$[A] は A から距離 0 の LAN すなわち A 自身の LAN である。$Ring_1$[A] は距離 1、すなわち 1 ホップ離れた LAN の集合である、以下同様である。

ERB 探索は次のように動作する。プロセッサ A がオブジェクト X の位置特定が必要であるとする。$i = 0$ から始めて、要求メッセージは $Ring_i$[A] のすべての LAN にブロードキャストされる。もし応答を受け取るならば、探索は成功でおわる。そうでなければ、タイムアウト期間が経過したのちに、i が 1 増加され要求のブロードキャストは繰り返される。リングサイズ I は、インターネットワークの直径によって概略的に制限される。

この方法は、最も近い複製の位置を提供するが、あるオブジェクトのすべての複製の位置を必ずしも同時に提供はしない。さらには、オブジェクト位置特定操作の効率は、位置特定時点でのクライアントからのオブジェクトの距離に直接的に比例する。

10.5.3　UID の中にオブジェクト位置を符号化による方法

この方式は構造化オブジェクト識別子を使用する。この方式での UID の 1 つのフィールドはオブジェクトの位置である。UID が与えられると、システムは、構造化 UID の適切なフィールドを調べることによって、その UID から対応するオブジェクトの位置を抽出する。抽出された位置はオブジェクトが存在するノードである。

これは直接的であり、効率的な方式である。しかし、この方式の 1 つの制限は、オブジェクトが一度ノードに割り付けられるとオブジェクトは移動を許されないということである。その理由は移動は識別子が変化することを要するからである。その結果として、あるオブジェクトはその生存期間の間、1 つのノードに固定される。この方式の別の制限は、1 つのオブジェクトの複数の複製をどのようにサポートするかが明確ではない。それゆえ、このオブジェクト位置特定方式の使用は、オブジェクト移動及びオブジェクト複製をサポートしない分散システムに制限される。

10.5.4 生成者ノードを最初に探索後ブロードキャストによる方法

この方式は前述の方式の単純な拡張である。拡張はオブジェクト移動機能をサポートするためである。オブジェクトは生成されたノードに留まることが多い (必ずしも真ではないが) という仮定に、この方法は基づいている。これは、オブジェクトの移動のコストが大きく、余り頻繁に移動することはないという理由である。

この方式において、構造化 UID の位置フィールドはそのオブジェクトが生成されたノードの識別子を含んでいる。UID が与えられると、生成者ノード情報が UID から抽出され、要求がそのノードに送られる。もしオブジェクトが既に生成者ノードに存在していないならば、クライアントノードに失敗の応答が返される。失敗の場合には、クライアントノードから要求をブロードキャストすることにより位置特定される。このオブジェクト位置特定方法は Cronus[Schantz 他 1986] において使用された。

ブロードキャストによる方法と比べて、この方法はネットワークトラフィックを大幅に減少させるのに役立つ。さらには、この方法は、オブジェクトの移動をサポートしているので、前述の UID にオブジェクト位置をコード化する方法よりも柔軟である。しかし、生成者ノードにおいて見つからないオブジェクトを位置特定するために使用するブロードキャストによって、この機構の規模拡張性は制限される。

10.5.5 転送位置ポインタによる方法

この方式は前述の方式の拡張である。この拡張の目標はブロードキャストプロトコルの使用を避けることである。**転送位置ポインタ (forward location pointer)** はオブジェクトの新しい位置を示すために使用される参照情報である。オブジェクトがあるノードから別のノードに移動させられるとき常に、転送位置ポインタが前のノードに残される。それゆえ、あるオブジェクトの位置を決めるためには、システムは最初にそのオブジェクトの生成者ノードにコンタクトし、それから単純に現在そのオブジェクトが存在しているノードへの転送位置ポインタに従うか又はポインタのチェーンに従う。

この方法はブロードキャストプロトコルを全面的に避けるという利点とオブジェクト移動機能をサポートするという利点をもつ。しかし、次の主な欠点を

実際にはもっている。

1. オブジェクト位置特定コストは、横断していくポインタのチェーンの長さに直接的に比例するし、チェーンが長くなるに従いコストは相当大きくなる。

2. ノードの故障により中間のポインタが紛失又は使用不可能となる場合にオブジェクトの位置特定は困難又は不可能とさえなる。それゆえ、この機構の信頼性は低い。

3. この方法は、維持のための付加的なオーバヘッドを招く。

10.5.6　ヒントキャッシュ使用とブロードキャストによる方法

もう 1 つのよく使用される方法は**キャッシュブロードキャスト (cache-broadcast)** 方式である。この方法では、各ノードに、一定数の最近参照した遠隔オブジェクトの **(UID, last known location)((UID、最後に知られている位置))** 対を含むキャッシュが維持される。UID が与えられると、ローカルキャッシュに、その UID をもつエントリがあるか否か調べる。もしエントリが見つかると、対応する位置情報がキャッシュから抽出される。オブジェクトアクセス要求は、抽出された位置情報で指定されるノードに送られる。しかし、もしオブジェクトがもはやそのノードに存在しないならば、要求はキャッシュから抽出された位置情報は古かったということを示す否定的な応答で返される。もし指定された UID がローカルキャッシュに見つからないならば、又はローカルキャッシュにおけるオブジェクトの位置情報が古いことがわかったならば、そのオブジェクトの現在位置を要求するメッセージがブロードキャストされる。このブロードキャストメッセージを受信する各ノードは指定されたオブジェクトを内部的に探索する。オブジェクトが見つけられたノードからクライアントノードに応答メッセージが返される。オブジェクトのこの位置がクライアントノードのキャッシュに記録される。キャッシュのエントリは、必ずしも正しくないので、ヒントとしてのみ提供される。

この方式は、あるノードからオブジェクトの位置特定を行う際に、高度のローカル性があるならば、非常に効率的な方法となり得る。また、オブジェクト移動機能をサポートできるので、柔軟性もある。さらには、キャッシュ情報の使用中更新の方法は、あるオブジェクトが移動するとき他のノードに知らせ

るコストと遅れを避けることができる。しかし、この方法の1つの問題は、ブロードキャスト要求がネットワークを散乱させ、たとえただ1つのノードが各オブジェクト位置特定操作に関っているとしても、すべてのノードを煩わす。

この方法は、最近の分散オペレーティングシステムにおいて最も一般的に使用されているオブジェクト位置特定機構である。これはAmoeba[Mullender 他 1990]、V-System[Cheriton 及び Mann 1989]、DOMAIN[Leach 他 1983]、NEXUS[Tripathi 他 1987]、Mach[Rashid 1987]、及び Chorus[Rozier 他 1988]において使用されている。

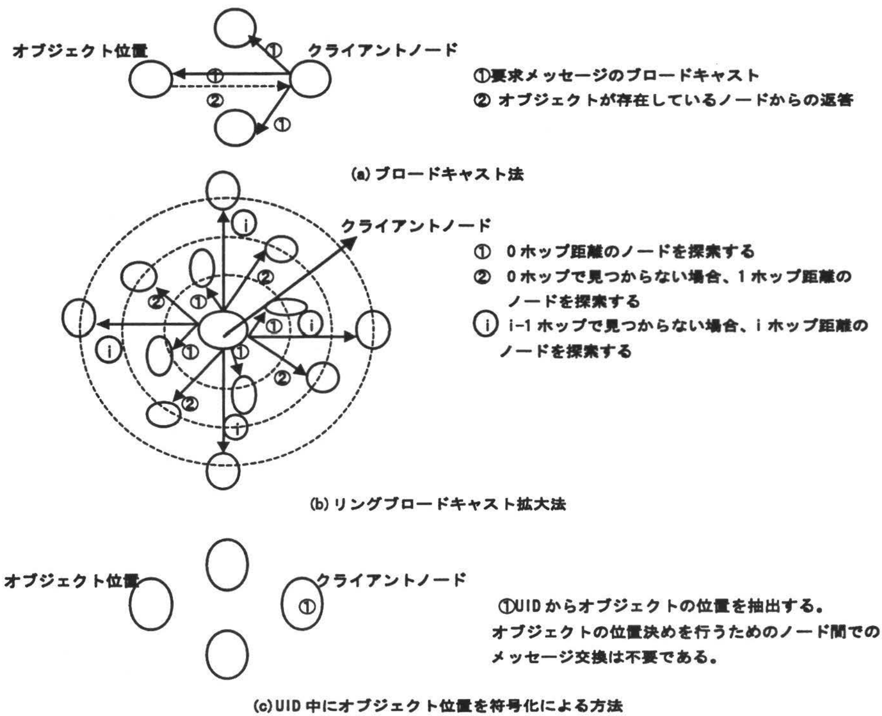


図 10.9　オブジェクト位置決め機構

図10.9と図10.10に、以上で論じたオブジェクト位置特定機構を要約している。方式(c)、(d)、及び(e)は構造化IDを要するが、方式(a)、(b)、及び(f)は非構造化及び構造化のいずれでも動作できることに注意されたい。また方式(c)のみが、ほかのどのノードともメッセージの交換を行うことなく、オブジェクト位置特定操作を実行できる。しかしこれは、オブジェクト移動とオブジェ

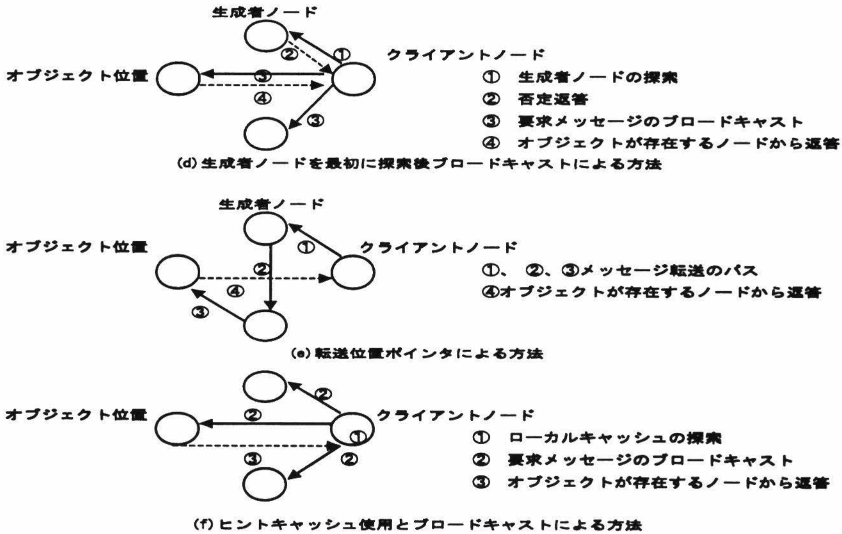


図 **10.10**　オブジェクト位置決め機構 (continued)

クト複製機能をサポートしていないので、柔軟性に乏しい。

10.6　人間指向名前

31A5 2B5F AD19 B1C8 のようなシステム指向名前は機械が扱うには有用であるが、ユーザの使用にとって適切ではない。ユーザはこれらの名前を記憶したり、タイプしたりすることを要求されると辛い。さらには、各オブジェクトはただ 1 つのシステム指向名前をもっているので、オブジェクトを共有するすべてのユーザはそのただ 1 つの名前を記憶し使用しなければならない。これらの制限を克服するために、ほとんどすべての名前付けシステムは、システムの様々なオブジェクトに対しユーザ自身が適切な名前を定義できる機能を提供している。システム指向名前の名前空間の直接上に名前空間を形成するこれらのユーザ定義のオブジェクト名前は**人間指向名前 (human-oriented name)** と呼ばれる。

人間指向名前は通常次の特性を持っている。

1. 人間指向名前はそれらのユーザに意味のある文字列である。

2. 人間指向名前はユーザによって定義され、使用される。

3. 1つの共有のオブジェクトに対し、異なるユーザは彼ら自身の適切な名前を定義し、使用することができる。すなわち、別名を付ける機能がユーザに提供される。

4. 人間指向名前は異なるオブジェクトに対する名前において可変長であるばかりではなく、同じオブジェクトに対する異なる名前においてさえ可変長である。

5. 別名機能により、2つの異なるオブジェクトを参照するのに同時に2人の異なるユーザによって同じ名前が使用されることもあり得る。さらには、一人のユーザが異なる瞬間で、異なるオブジェクトを参照するのに同じ名前を使用することもあり得る。それゆえ、人間指向名前は空間的にも時間的にもユニークではない。

名前空間の容易で効率的な管理の利点の理由で、階層的分割名前空間が人間指向オブジェクト名前によく使用される。階層的名前付けコンベンションが使用されるとき、一定のレベル数を使用するのか、任意のレベル数を使用するのかを決定することは重要である。すなわち、もし階層的名前付けシステムの名前が $c_1/c_2/.../c_i$ の形式であるならば、i は一定であるべきか任意であるべきか。両方式ともに利点、欠点をもっている。一定レベル数方式の主な利点は、この方式は、任意レベル数方式に比べて実装が単純で容易であるという点である。これは、任意レベル数方式におけるすべてのソフトウェアは任意の数のレベルを扱えなければならず、名前を扱うソフトウェアは一定レベル数方式のソフトウェアよりも複雑になりがちである。一定レベル数方式の欠点は、レベル数を決定するのが難しいという点である。もし新しいレベルが後に追加されるならば、新しいレベルを考慮するために名前処理のアルゴリズムを変更しなければならず、かなりの作業が発生する。

他方、任意レベル数方式は、独立に存在する名前空間を組み合わせて1つの新しい名前空間に容易に拡張できるという利点をもっている。例えば、図 10.11 に示すように、2つの独立の名前空間は、既存の2つのルートを子供にして、新しいルートを加えることにより、1つの名前空間に組み合わせることができる。主な利点は、もしある名前が旧名前空間で明確であったなら、たとえ組み合わされた他の名前空間に同じ名前が出現したとしても、新名前空間においても明確である。

一定レベル数方式よりも柔軟性の高い任意レベル数方式は、最近のほとんど

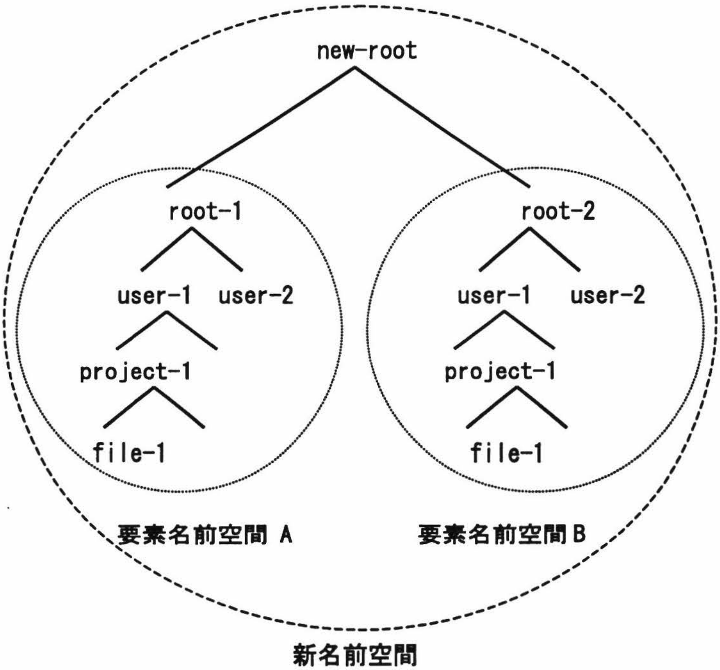


図 10.11　新しいルートを付加することにより、2 つの名前空間を組み合わせて 1 つの名前空間を形成する。

の分散システムによって使用されている。それゆえ、本章の以下では、任意レベル数階層的名前空間を扱う。人間指向名前に関する主な問題には以下の事項がある。

1. 全体 (グローバル) のオブジェクト名前付けに適切な方式を選択すること
2. 名前空間をコンテキストに分割する適切な方式を選択すること
3. コンテキスト結び付けの実装に適切な方式を選択すること
4. 名前リゾルーションに適切な方式を選択すること

これらの問題を次節に述べる。

10.6.1 人間指向階層的名前付け方式

分散システムにおける様々なオブジェクトにシステム全体でユニークな人間指向名前を割り付けるために4つの方法がある。

オブジェクトのローカル名前とホスト名前とを組み合わせる方法

この名前付け方式は、幾つかの**分離名前空間 (isolated name space)** を含む1つの名前空間を使用する。各分離名前空間は分散システムの1つのノードに対応し、この名前空間の名前はユニークにノードの1つのオブジェクトを特定する。全体システムにおいて、オブジェクトはホスト名前とローカル名前の何らかの組み合わせによって名前付けされ、それがシステム全体でのユニーク名前を保証する。例えば、Ibis[Tichy 及び Ruan 1984] において、ファイルオブジェクトは **host-name : local-name** の形式をもつ名前によってユニークに特定される。ここで、**host-name** はファイルが存在するノードの識別子であり、**local-name** はそのホストノード上のファイルをユニークに特定する UNIX 類似の**絶対パス名前**である。

この名前付け方式は単純で実装が容易である。しかし、この方式の主な欠点は、名前は位置透過でもなく、位置独立でもない点である。さらには、あるオブジェクトがあるノードから別のノードへの移動ごとに、オブジェクトの絶対名前が変化するという意味で柔軟性に乏しい。それゆえ、最近の分散システムに適していない。

分離名前空間を単一名前空間にリンクする方法

この方式では、全体の名前空間は幾つかの分離名前空間を含む。しかし、分離名前空間が分離したまま残る前述の方法とは異なり、単一の名前付け構造を形成するために、分離名前空間は一緒に結合される。これらの要素名前空間の名前付け階層における位置は任意である。名前付け構造において、要素の名前空間は、他の要素名前空間の直接的下に、又は他の要素名前空間を通した位置に置かれるかもしれない。この方式において、絶対パス名前 (absolute pathname) という概念はない。各パス名前は、現在動作中のコンテキスト、又は現在の要素名前空間のどれかのコンテキストに相対的である。

この名前付け方式は UNIX United[Brownbridge 他 1982] において使用されている。ある UNIX United システムは、この名前付け方式を用いて任意の数の UNIX システムを**結合 (join)** している。UNIX United の単一名前付け構造

において、各要素システムはあるノードに所属する完全なUNIXディレクトリツリーである。各要素名前空間のルートは、外部からアクセスできるように、また区別できるように、ユニーク名前が割り付けられる。ある要素自体のルートは/として参照される。また/で始まるすべてのパス名前の開始点としても働く。UNIXの表記法../は要素自体のルートの親を参照するために使用される。それゆえ、それ自体の親であり、文字列の名前を割り付けられていないただ1つのルート、すなわち複合構造のルートが存在する。これは全体の構造を1つのツリーとするために必要な仮想的なルートである。

そのような名前付け構造の単純な例を図10.11に示す。そこでは、複合名前構造は2つのそれぞれのルートが名前root-1及びroot-2である要素名前空間A及びBからなる。この方式における相対パス名前付けの説明のように、要素名前空間Aの中から、2つの要素名前空間A及びBのfile-1は、それぞれ/user-1/project-1/file-1及び ../root-2/user-1/project-1/file-1で参照される。同様に、もしクライアントの現在の動作中のディレクトリが、要素名前空間Bのproject-1ならば、2つの要素名前空間A及びBのfile-1は、それぞれ../../../root-1/user-1/project-1/file-1及びfile-1で参照される。

この名前付けシステムの主な利点は、既存の名前空間を単一の全体名前空間に結合することが簡単であるという点である。しかし、全体オブジェクト名前付けにこの方式を使用する名前付けシステムにおいて、扱わねばならない1つの重要事項は、クライアントが分離名前空間において使用していた名前を、新しい名前空間構造に適合するように修正することなく、継続して使用可能とすることである。例えば、図10.11において/user-1/project-1/file-1は2つの名前空間が統合される前にクライアントによって使用されていた名前である。統合前は、この名前は要素名前空間Aのクライアントに対して要素名前空間Aのfile-1を参照し、要素名前空間Bのクライアントに対して要素名前空間Bのfile-1を参照する。それゆえ、2つの要素名前空間の統合の後にも同様に、2つの名前空間のクライアントに対してそれぞれ同じオブジェクトを参照することが望ましい。DECシステム研究センタにおいてLampson及びその協力者によって設計及び実装されたGlobal Name Service(GNS)[Lampson 1986]はこの問題に対して興味ある解決法を提供している。**ディレクトリ識別子(DI：directory identifier)**と既知のディレクトリのテーブルの使用に基づくこの方法を以下に述べる。

GNSは2つの部分⟨*directoryname, valuename*⟩をもつ。最初の部分はディレクトリを特定し、第2の部分は値のツリー(value tree)又は値のツリーの一

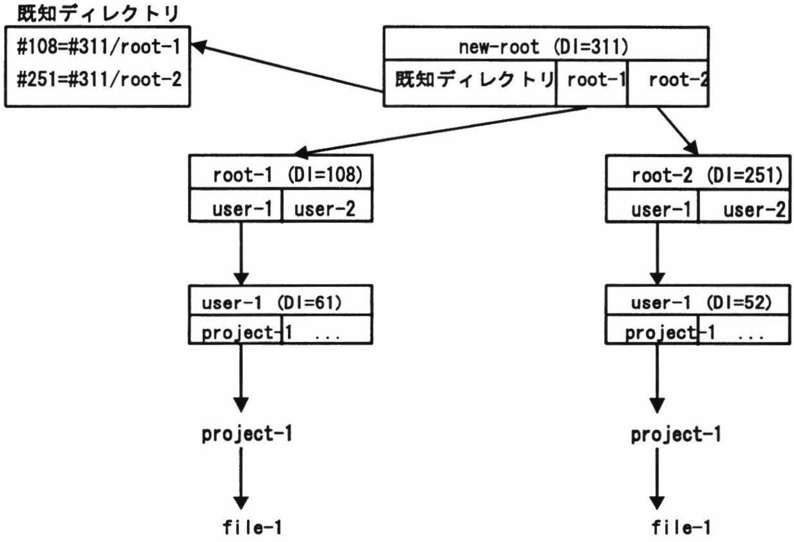
部を参照する。各ディレクトリはユニーク DI をもっている。図 10.11 は GNS 名前付け方式を図示するために 10.11(a) に描き直されている。

GNS 名前付けコンベンションにおいて、統合前の要素名前空間 A 及び B の file-1 は</user-1, project-1/file-1 >として参照されるであろう。2 つの名前空間の統合の後では、要素名前空間 A のクライアントがこの名前を使用するとき、ローカルルートディレクトリの DI を知っているローカル名前エージェントは DI(#108) を前置きし、名前<#108/user-1, project-1/file-1 >を生成する。他方、要素名前空間 B のクライアントが同じ名前を使用するとき、ローカル名前エージェントは、要素名前空間 B のルートディレクトリの DI(#251) を前置きし、名前<#251/user-1, project-1/file-1 >を生成する。名前エージェントはこの導出された名前を GNS サーバにルックアップ要求として渡す。相対名前は GNS サーバに渡す前にローカル名前エージェントによってローカル絶対名前に変換される。

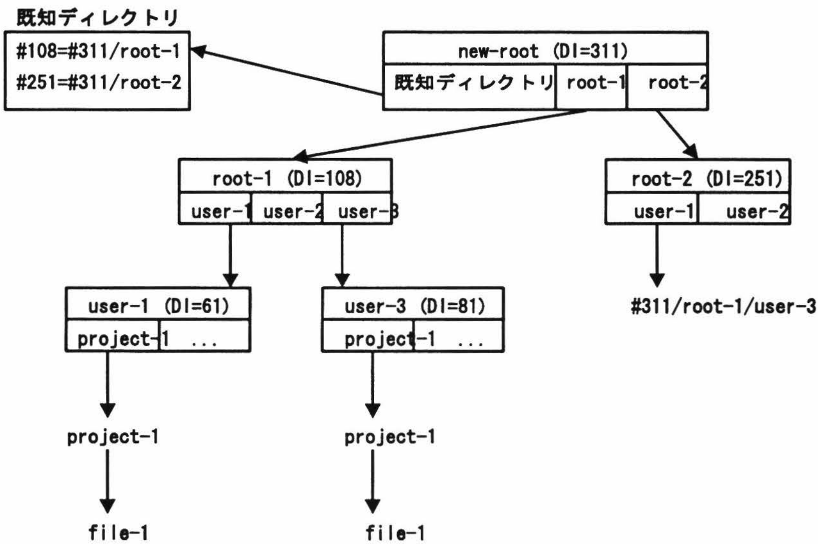
GNS サーバが、DI(#108 のように) が与えられたディレクトリの位置特定ができるように、GNS は既知ディレクトリのテーブルを使用する。このテーブルはすべての要素名前空間のルートディレクトリのリストをもっている。2 つ以上の名前空間を統合することにより生成された名前空間は、その後いつか、別の名前空間と統合され、新しい名前空間を形成するかもしれないので、このテーブルは、基本的に全体の名前空間内で動作中ルートとして使用されるすべてのディレクトリをもつ。図 10.12(a) の例において、2 つの動作中ルート (root-1 及び root-2) のみであるので、このテーブルは 2 つのエントリをもつ。各 GNS サーバはこのテーブルのコピーをもつ。

名前空間の実際のルートが変更になると常に、すべての GNS サーバは実際のルートの新しい位置を知らされる。実際のルートで始まる名前 (< new-root/root-1/user-1, project-1/file-1 >のような) は通常の方法で GNS サーバによって解釈される。他方、動作中ルートの DI で始まる名前 (<#108/user-1, project-1/file-1 >のような) は既知ディレクトリのテーブルから対応する DI のエントリに対する情報を使用することによって解釈される。それゆえ、GNS 方式は絶対名前及び動作中ルートに相対的な名前の両方が使用可能である。

組織上の変更に対応するための名前空間の再構築も GNS はサポートする。例えば、図 10.11 の 2 つの名前空間 A 及び B は 2 つの異なる会社を表し、それらが統合されて、1 つの会社の 2 つの支社になる場合を仮定する。そしてある時間の後に、要素名前空間 B に対応する支社の user-1 が、要素名前空間 A に対応する支社に転任になる場合を想定する。この変化に対応するために、まし



(a)



(b)

図 10.12　存在する名前空間における変化に対応する GNS における方法、(a)2 つの名前空間を組み合わせて 1 つの名前空間を形成する；(b) 名前空間を再構成する。

user-1 のサブツリーが単純に root-1 ディレクトリに移動されるなら、次のような問題が生じる。

1. user-1 は既に root-1 の下に存在する。この問題は転任されたユーザに、例えば user-3 といった新しいユーザ識別子を割り付けることで、容易に解決できる。

2. ＜ new-root/root-2/user-1, ... ＞の形式の絶対名前はもはや動作せず、変更が必要である。GNS はこの問題を、図 10.12(b) に示すように、「シンボリックリンク」を root-2 ディレクトリにおける user-1 に対する元のエントリの場所に挿入することにより解決している。GNS ディレクトリ探索手続きは、このリンクをこのユーザの新しいディレクトリ位置への振り向け先 (redirection) として解釈する。

明示的要求による遠隔名前空間の共有

サンマイクロシステムズ (Sun Microsystems) 社の Network File System(NFS)[Sandberg 他 1985] によって知られるこの方式もまた、新しい名前空間を生成するために、様々なノードの分離名前空間をアタッチするという考えに基づく。しかし、システムによって単一の名前付け構造が生成される前述の方式とは異なり、この方式では、ユーザは遠隔名前空間のコンテキストを彼ら自身のローカル名前空間にアタッチする柔軟性を与えられている。遠隔コンテキストが一度ローカルにアタッチされると、遠隔のオブジェクトを位置透過的な方法で名前付けすることができる。目標は、ユーザによる明示的な要求によって、透過的な方法で、多数の名前空間の間で、ある程度まで共有することを許すことである。それゆえ、結果としての名前構造の全体としてのビューは、ノードごとにもつツリーの森 **(forest)** であり、一部共有サブツリーが存在するので重複している。各ノードの名前空間の全体が単一の名前付け構造にアタッチされる前述の方式とは異なり、この方式では、ユーザは、彼らのローカル名前空間に完全な遠隔名前空間ツリー又はその一部 (サブツリー) をアタッチする柔軟性をもっていることに注意されたい。また遠隔名前空間を共有する要求は、その要求が行われたノードにのみ影響を与え、他のノードには影響を与えないことに注意されたい。これがノードの独立性を保証する。NFS におけるファイルオブジェクトの名前付けをこの方式の説明として以下に述べる。

NFS において、マウント **(mount)** プロトコルは遠隔名前空間ディレクトリ

をローカル名前空間にアタッチするために使用される。それゆえ、遠隔ディレクトリをあるノードから透過的にアクセス可能とするために、そのノードからマウント操作が行われなければならない。マウント操作を行うノードはクライアントノードと呼ばれ、名前空間ディレクトリがマウントされるノードはサーバノードと呼ばれる。NFS ではそれぞれのマシンが同時にクライアントにも、サーバにもなれる。サーバはその名前空間ツリーの、遠隔クライアントからアクセスされるすべてのディレクトリを**エクスポート**しなければならない。エクスポートされたディレクトリをルートとする完全なサブツリーは 1 つのまとまりとしてエクスポートされる。それゆえ、エクスポートされたディレクトリがクライアントのローカル名前空間にマウントされるとき、エクスポートされたディレクトリの下のすべてのディレクトリは自動的にクライアントにとってアクセス可能となる。これは、クライアントがサーバのディレクトリをマウントするとき、マウント処理がクライアントのローカル名前クライアントのマウント点とサーバの名前名前空間のエクスポートされたディレクトリとの間にリンクを設定するからである。ディレクトリがサーバが立ち上がり時にサーバによって自動的にエクスポートされるために、サーバによってエクスポートされるべきディレクトリのリストが /etc/exports ファイルに維持される。

マウント操作のセマンティクスは、あるサーバからエクスポートされるサーバのファイルシステムディレクトリが、クライアントのファイルシステムのディレクトリの上にマウントされるということである。マウントされたサーバファイルシステムのディレクトリのサブツリーは、クライアントのファイルシステムの統合された一部として現れる。そして、サーバのディレクトリがマウントされた位置のクライアントのローカルディレクトリから下のサブツリーを置き換える。ローカルディレクトリは新しくマウントされたディレクトリのルートの名前になる。マウント操作の後に、ローカルユーザは全体として透過的な方法でマウントされた遠隔ディレクトリのファイルをアクセスできる。クライアントのマシン上で動作しているプログラムに対して、遠隔ファイルサーバに存在するファイルとローカルディスクに存在するファイルとの間に (ほとんど) 差はない。

クライアントは次の方法のいずれかの 1 つでディレクトリをマウントできる。

1. **手動マウンティング (manual mounting)**:この場合には、サーバのディレクトリがクライアントのローカル名前空間にマウントされる必要な時点で、クライアントは **mount** コマンドを使用する。 クライアントがサーバのディ

レクトリのサブツリーにあるファイルをアクセスする必要がなくなったとき、**unmount** コマンドにより、クライアントがそのディレクトリをアンマウントすることができる。クライアントは **mount** 及び **unmount** コマンドを使用できるスーパーユーザでなければならない。この方法は必要に応じて、サーバのディレクトリを動的にマウントしたりアンマウントしたりする柔軟性をクライアントに提供する。

2. **静的マウンティング (static mounting)**:手動マウンティング方式において、サーバディレクトリのマウントが必要になるたびに、**mount** コマンドがユーザによって使用されなければならない。静的マウンティング方式によると、手動の介入なしに、サーバによってエクスポートされたディレクトリからクライアントの選択に応じたディレクトリをクライアントが自動的にマウントできる。この方法では、クライアントマシン上の **/etc/rc** と呼ばれるファイルの中に、指定のディレクトリをマウントするためのコマンドに対するシェルスクリプトが書かれる。このシェルスクリプトはクライアントマシンが立ち上がるときに自動的に実行される。それゆえ、すべてのマウント動作は立ち上がり時に自動的になされる。

3. **自動マウンティング (automounting)**:静的マウンティング方式において、マウント動作は立ち上がり時になされるので、クライアントの /etc/rc ファイルにリストされたサーバのディレクトリが、クライアントによって使用されないときでも、すべてマウントされる。自動マウンティング方式によると、サーバのディレクトリはクライアントが必要なときのみマウントされるようにできる。すなわち、この方式では、サーバのディレクトリは必要に応じて自動的にマウント及びアンマウントされる。

この方式では、サーバのディレクトリはクライアントのローカルディレクトリと関連付けられるが、クライアントがサーバのディレクトリの下の名前空間にありファイルをアクセスするためのコマンドを起動するときのみ、実際のマウンティングが行われる。一度マウントされると、サーバのディレクトリは、それが必要である限りマウントされたままとなる。サーバのディレクトリの下の名前空間がクライアントによって5分間アクセスされないと、サーバのディレクトリは自動的にアンマウントされる。

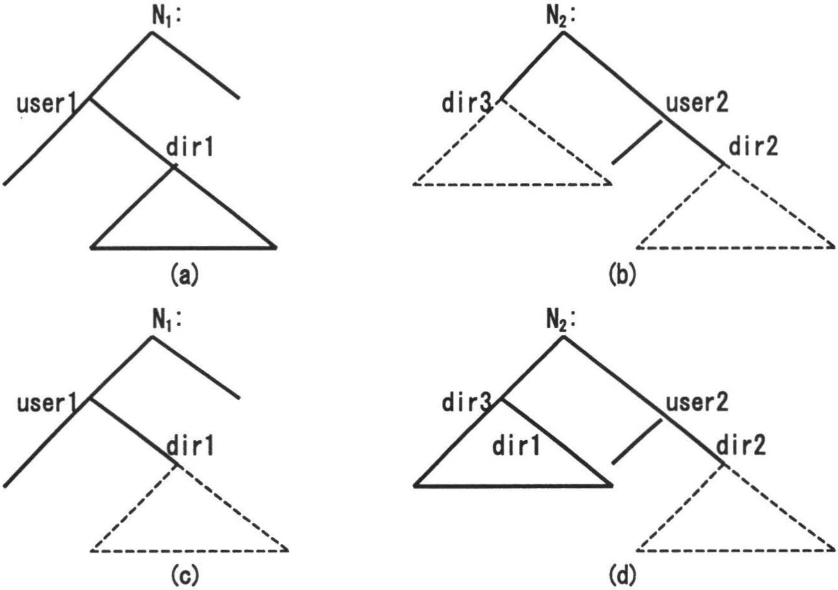
この方式の他の特徴は、クライアントのローカル名前空間に、1つのサーバのディレクトリの代わりに1式のサーバのディレクトリが同時に関連付けられ

るかもしれない。この場合には、クライアントがマウントされたディレクトリの下の名前空間のファイルを最初にアクセスするとき、オペレーティングシステムが、ディレクトリが関連付けられたサーバのそれぞれにメッセージを送信する。応答した最初のサーバが勝ち残りそのディレクトリがマウントされる。

自動マウンティング方式は、クライアントが実際に必要とするときのみサーバが動作していればよいので、フォールトトレラント性を達成する助けとなる。さらには、1 つのサーバの代わりに、1 式のサーバが関連付けられるので、ただ 1 つのサーバが生きていればよい。

NFS のファイルマウンティング機構を説明するために、[Silberschatz 及び Galvin 1994] によって与えられた例に基づく図 10.13 の例を考える。図 10.13(a) 及び (b) はそれぞれノード N_1 及び N_2 の名前ツリーの、説明に関係する部分を示す。三角形はディレクトリのサブツリーを表す。マウンティングがまだ行われていないので、各ノードではローカルファイルのみをアクセスできる。図 10.13(c) は、ノード N_2 のディレクトリ /user2/dir2 をノード N_1 のディレクトリ /user1/dir1 にマウントした後のノード N_1 の関係する部分の名前ツリーの構造を示す。このマウンティングの後、dir2 は共有ディレクトリとなり、ノード N_1 及び N_2 のユーザによって共有される。ノード N_1 上のユーザは、プレフィックス/user1/dir1/dir2 を用いることにより dir2 にあるどのファイルもアクセスすることができる。ノード N_1 上のディレクトリ/user1/dir1 の下の元々のサブツリーはもはや透過的となる。

この方式の利点は、ユーザが自己のローカルノードの名前空間の上に、遠隔名前空間の必要な部分だけをアタッチできる柔軟性を提供する。この方式の別の利点は、UNIX への修正として実装が簡単であるという点である。しかし、幾つかの欠点もある。主な欠点は、すべてのノードに対して一貫性のある全体名前空間をこの方式は提供しないことである。したがって、同じオブジェクトは、異なるノードからみるとき、異なる「絶対」名前をもつかもしれない。図 10.13(c) 及び (d) はそれぞれユーザの要求によってマウンティングが行われた後の、ノード N_1 及び N_2 の名前ツリー構造を示す。ディレクトリ dir2 におけるオブジェクトの絶対名前は、ノード N_1 及び N_2 において、それぞれプレフィックス/user1/dir1/dir2 及びプレフィックス/user2/dir2 をもつ。この絶対名前機能の不足によって、異なるノードで動作しているプロセスは異なる名前付けドメインの中になるので、分散アプリケーションプログラムを作成するのが困難となることがある。例えば、データファイルを



(a)ノードN1の名前ツリーの一部

(b)ノードN2の名前ツリーの一部

(c)dir1にノードN2のdir2をマウントした後のノードN1の名前ツリーの構造

(d)dir3にノードN1のdir1をマウントした後のノードN2の名前ツリーの構造

図 **10.13** NFS名前付け方式のファイルマウント機構

処理するために、幾つかのノードを使用するアプリケーションは、もしファイル名前が/user1/project1/data1として指定されていると、問題があるであろう。すなわち、同じファイルをオープンする代わりに、識別子rome及びparisをもつノード上のプロセスは、それぞれ/rome/user1/project1/data1及び/paris/user1/project1/data1オープンするであろう。たとえもしユーザが明示的に/paris/user1/project1/data1と指定しても、もしromeがparisのファイルシステムをマウントしていないならば、さらに悪く、名前/parisの下に他のノードのファイルシステムがマウントされていると、プログラムは誤動作する。すべてのノードが同じオブジェクトに対して同じ名前を使用することを保証するただ1つの方法は、各ノードのマウントテーブルを注意深く人間が管理することである。この方法を用いるシステムは規模拡張性に乏しい。その理由

は、もしすべてのノードが他のすべてをマウントするならば、システムにおけるマウント個所の数は n^2 に比例する ($n =$ ノードの総数) ので、管理及び計算のオーバヘッドが大きくなる。

この方式の別の重大な欠点は管理上の複雑性である。故障したノードの影響又はノードをオフラインにする影響により、異なるノード上の、ある任意のディレクトリの集合が使用できなくなる。例えば、オブジェクトを 1 つのノードから別に移動させることは、すべての関連するノードの名前空間の変更を要する。さらには、マウント操作の引数による遠隔ディレクトリの指定は非透過的な方法でなされるので、この方式は完全な透過性を提供していない。この引数は遠隔ディレクトリの位置 (すなわちノード識別子) が指定されなければならない。位置独立性もサポートされていない。その理由は、上に論じたように、あるオブジェクトの絶対名前は、オブジェクトがアクセスされるノードに依存するからである。

単一のグローバル名前空間

LOCUS[Walker 他 1983]、Spite[Welch and Ousterhout 1986]、V-System [Cheriton 及び Mann 1989]、及び Galaxy[Sinha 他 1991] のような幾つかの分散オペレーティングシステムは、オブジェクトの名前付けに単一のグローバル名前空間 (single global name space) の方式を使用している。この方式では、単一グローバル名前構造はシステムにおけるすべてのノードのすべてのオブジェクトを覆っている。したがって、同じ名前空間がすべてのユーザにとって可視であり、オブジェクトの絶対名前は、オブジェクトの位置又はオブジェクトをアクセスするユーザの位置に関係なく、常に同じである。それゆえ、この方式は位置独立性の両方のタイプ、すなわちオブジェクトをアクセスしている側 (例えばプロセス) の位置独立性及びアクセスされているオブジェクトの位置独立性をサポートしている。ユーザの移動性の特性は、すなわちオブジェクトをアクセスする側の位置独立性は、NFS ではネットワークのすべてのマシンのユーザのホームディレクトリの上に共有ファイルシステムをマウントするマウント機構によっても提供されている。しかし、NFS においてすべてのユーザにこの特性をサポートすることは、相当な管理及び計算のオーバヘッドを必要とする。

単一グローバル名前空間の方式は、完全な位置透過性及び位置独立性という特性のゆえに、最近の分散オペレーティングシステムにおいて、最もよく使用されている方式である。それゆえ、本章の以下の節では、この名前付け方式に

絞って述べる。

10.6.2 名前空間をコンテキストに分割

名前空間管理は、オブジェクト名前、オブジェクト属性、及びそれら2つの結び付けからなる**名前付け情報 (naming information)** の格納と維持を含む。分散システムにおける信頼性及び空間オーバーヘッド問題のゆえに、完全な名前付け情報を集中したノードに格納すること、又はすべてのノードに複製をもたせることは望ましくはない。それゆえ、名前付け情報は通常分散され、複製される。すなわち、1つのグローバルな名前サーバの代わりに、多数の名前サーバが存在し、それぞれが完全名前付け情報の一部分の複製を格納している。これらの名前サーバは、名前付け情報を維持するため、及びユーザから要求された名前を変換するために、名前サーバの間で相互動作する。基本的な問題は、どのように名前付け情報を分割し、名前サーバの間に分散するかという点である。このための分割機構の主な目標は、名前付け情報の維持及び名前の変換に要するオーバーヘッドを最小にすることである。

コンテキスト (context) の概念が名前空間を小さい要素に分割するために使用される。コンテキストは、名前付けされたオブジェクトに関する情報の格納及び複製のための細分できない単位を表している。名前空間は**クラスタリング条件**を使用することにより、コンテキストに分割される。クラスタリング条件とは、ある名前に適用されるとき、与えられた名前がこのクラスタリング条件によって指定されたコンテキストの中に存在しているか否かによって、真又は偽の値を返す表現式である。3つの基本的なクラスタリング方法は、アルゴリズム的クラスタリング、シンタクス的クラスタリング (syntactic clustering)、及び属性クラスタリングである。以下にこれらを述べる。

アルゴリズム的クラスタリング

この方法では、名前にある関数を適用して得られた値により、名前がクラスタされる。それゆえ、**アルゴリズム的クラスタリング**において、クラスタリング条件は $function(name) = value$ という形式の述語である。例えば、名前をバケットにクラスタするハッシュ関数は、アルゴリズム的クラスタリングのクラスタリング条件の良い例である。

図10.14にアルゴリズム的クラスタリングの簡単な例を示す。この例において、最初に、関数 func-1 が名前空間を2つのコンテキストに分割するために、

名前に適用される。2つのコンテキストのうちの1つはまだ大き過ぎる。それゆえ、第2の関数 (func-2) が、より小さいコンテキストに分割するために、このコンテキストの名前に適用される。このように、単一のコンテキストとしての完全な名前空間からスタートして、適切な大きさのコンテキストのグループを生成するために一連のクラスタリング条件が適用される。

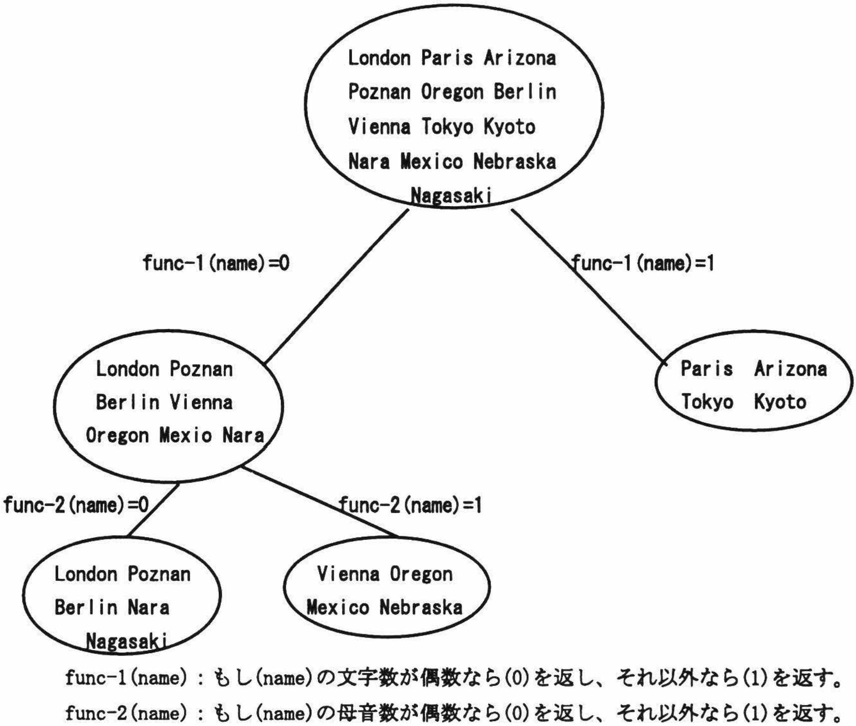


図 **10.14**　アルゴリズム的クラスタリングの例

アルゴリズム的クラスタリング機構の主な利点は、これは**構造フリー名前分散 (structure-free name distribution)** をサポートしていることである。構造フリー名前分散は、名前空間の部分への管理的制御に関して何ら制限を置かない方式である。このような名前空間の分割は、名前の大きさ、要素名前の数、要素名前の順番、又は名前の中の文字の種類など、すなわち名前の構造には対応しない。特に、あるオブジェクトのオーナは、オブジェクトの名前とは独立に、管理的制約にしたがって、権限名前サーバを選択するかもしれない。

これはあるオブジェクトに対して権限名前サーバの管理組織の割り当て(及び再割り当て)に関して最大の柔軟性を与える。アルゴリズム的クラスタリングはまた、フラット名前空間に対してさえ、健全な名前リゾルーションツリーを構築することを可能とする。しかし、この方法の主な欠点は適切なクラスタリング関数を設定することが困難であることである。これは多様なユーザによって定義される名前の特性が通常、予測不能であるからである。それゆえ、もしクラスタリング関数が最初に適切に設定されないと、他のコンテキストが非常に小さいのに、あるコンテキストは大きくなり過ぎるかもしれない。

シンタクス的クラスタリング

この方法は、名前空間をコンテキストに分割するために最もよく使用される方法である。**シンタクス的クラスタリング (syntactic clustering)** はパターンマッチング手法 **(pattern-matching technique)** を用いて行われる。**パターン (pattern)** は名前が比較されるテンプレートである。パターンには、単にアスタリスクによって表されどのような文字列ともマッチするワイルドカードを含むものから通常の表現まである [Terry 1986]。特定のパターンを意味するシンタクス的クラスタリング条件は、ある名前に適用されると、もし名前がそのパターンにマッチすると TRUE を戻す。共通プレフィックスをもつ名前 prefix *のようにある特定のパターンにマッチするすべての名前は同じコンテキストの部分と判断される。

説明のため、図 10.15 に名前のシンタクス的クラスタリングの 2 つの単純な例を示す。図 10.15(a) は構造化された名前の分割を示し、図 10.15(b) はフラット名前の分割を示す。図 10.15(a) から、階層的に構造化された名前空間におけるパターンマッチングは通常、要素ごとに行われ、通常、その結果のコンテキストはマッチしなかった名前の部分のみを含んでいる。それゆえ、次のレベルでは、新しいクラスタリング条件は名前の後続の部分に対してのみ適用される。しかし、シンタクス的クラスタリング条件は、各ステップにおける 1 つの追加要素のマッチングに限定されない。すなわちこれは可変数の要素が同時にマッチされる可変シンタクス的クラスタリングになる。

シンタクス的クラスタリング機構によると、事実上すべての名前管理シンタクスにおいてなされるように、名前はその構造に類似した方法で変換される。これは、単純なマッチングはクラスタリング技法として十分であることを意味する。

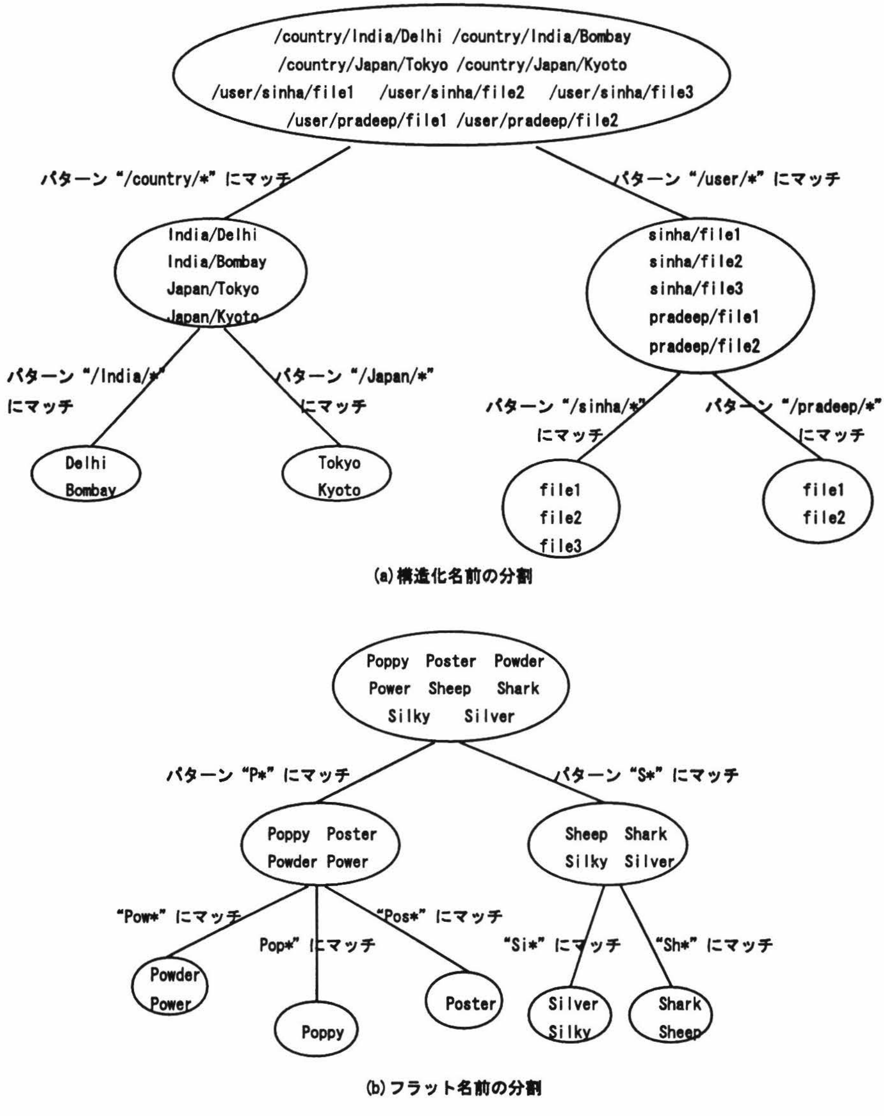


図 10.15　シンタクス的クラスタリングの例 :;

属性クラスタリング

属性クラスタリング (attribute clustering) の方法は属性ベースの名前付けコンベンションによって使用される。この方法では、名前によって所有される属性に基づき、名前がグループ化される。ある属性はタイプと値の両方をもっている。タイプは値フィールドのフォーマットと意味を示す。それゆえ、同じ属性 (タイプ、値) の対をもつすべての名前は、属性クラスタリング機構における同じ区分 (partition) に置かれる。

説明として、図 10.16 に名前の属性クラスタリングの簡単な例を示す。階層的に構造化された名前空間において、属性クラスタリングは通常、属性ごとに行われ、通常、その結果のコンテキストは名前のマッチしなかった属性のみを含んでいる。それゆえ、次のレベルにおいて、新しいクラスタリング条件が残っている名前の属性についてのみ適用される。しかし、属性クラスタリング条件は、各ステップにおける 1 つの追加属性のマッチングに限定されず、1 つのステップで複数の属性がマッチされるかもしれない。

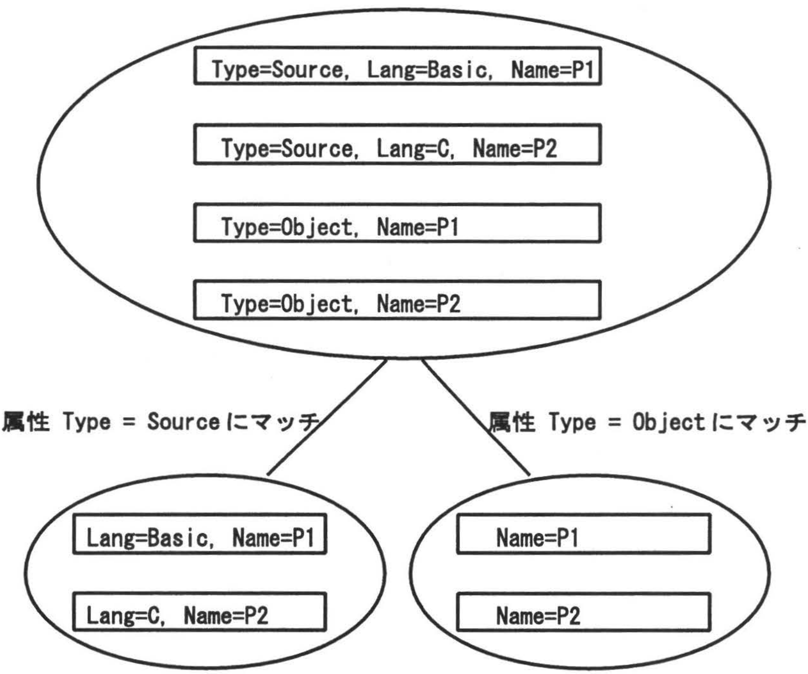


図 10.16 属性クラスタリングの例

10.6.3　コンテキスト結び付け

名前空間のコンテキストは、名前空間を管理している多くの名前サーバの間に分散される。すなわち、名前サーバは名前空間のコンテキストの集合の小さいサブセットのみを通常格納する。それゆえ、ある名前の変換が必要なとき、サーバは、名前付けされたオブジェクトの**権威属性 (authority attribute)** のためのローカルコンテキストを最初に探索する。権威属性は、名前付けされたオブジェクトに対する**権限名前サーバ (authoritative name server)** のリストを含んでいることを思い出されたい。もしそのオブジェクトの権威属性がローカルコンテキストに見つからないならば、**コンテキスト結び付け (context binding)** と呼ばれる追加の構成データが**名前リゾルーション (name resolution)** のためにローカルに存在しなければならない。コンテキスト結び付けは、格納されているコンテキストを他のコンテキストに関連付けさせる。すなわち、名前付けされたオブジェクト及びそのコンテキストが格納されている名前サーバについてさらに知識を与える。コンテキスト結び付けを実装するために通常使用される 2 つの方式は、テーブル準拠方式及び手続き準拠方式である。以下にそれらを述べる。

テーブル準拠方式

テーブル準拠方式 (table-based strategy) は、階層的なツリー構造の名前空間においてコンテキスト結び付けを実装するために最もよく使用される方法である。この方法において、各コンテキストは 2 つのフィールドをもつテーブルである。最初のフィールドは名前付けされたオブジェクトの要素名前を格納し、第 2 のフィールドはコンテキスト結び付け情報又は権威属性情報を格納する。名前の最後の要素に対応するエントリは権威属性情報を含む、一方、他のエントリはコンテキスト結び付けを含む。コンテキスト結び付けは、名前空間の管理部分に対する権威の代表を反映している。階層の様々なレベルでのコンテキストオブジェクトに格納されねばならない構成データの量は、名前空間ツリーの分岐の程度に比例している。テーブル準拠方式のコンテキストは**ディレクトリ (directory)** として知られている。

説明として、図 10.17 にコンテキスト結び付けを実装するためのテーブル準拠方式の例を示す。図はまた、この実装方式において、どのようにして、あるオブジェクトが 2 つの異なる人間指向名前をもつことができるのかを示す。それゆえ、共有オブジェクトのユーザは、そのアクセスのために、彼ら自身の適

切な名前を指定できる。

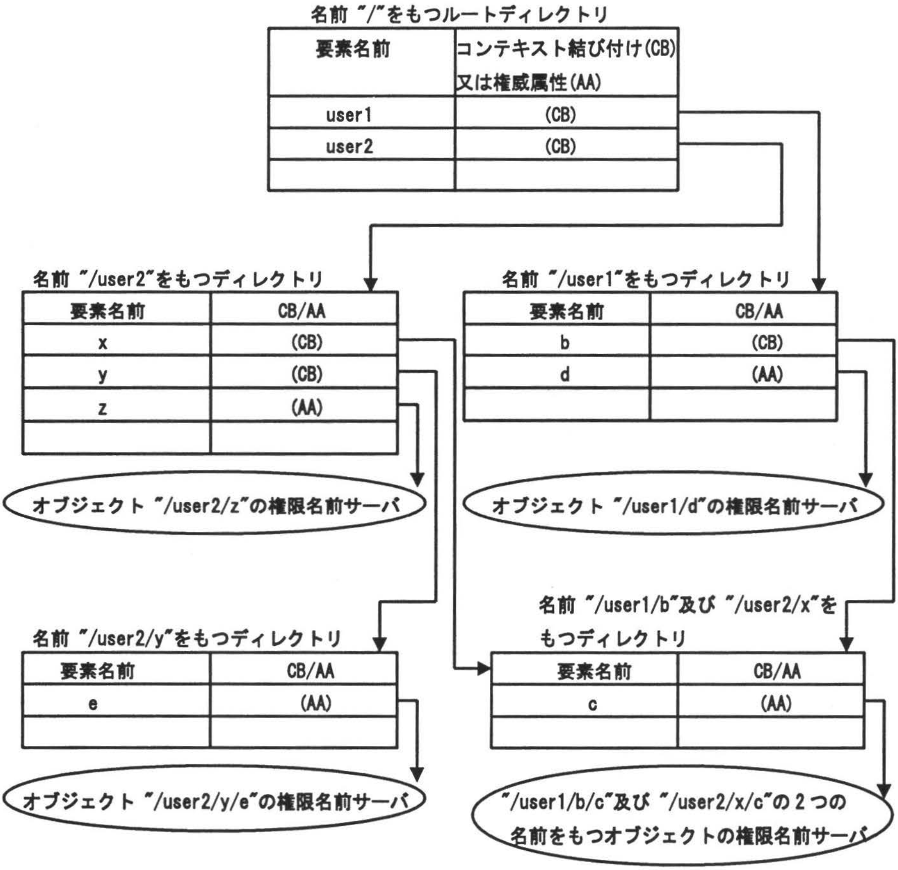


図 10.17 コンテキスト結び付けを実装するためのテーブルベース方式の例

手続き準拠方式

名前サーバは、与えられたコンテキストにおいて結び付けを実装するために、テーブル準拠方式ではなく、**手続き準拠方式 (procedure-based strategy)** を使用するかもしれない。この方法では、コンテキスト結び付けは手続き (procedure) の形をとる。この手続きは、実行されるとき、名前付けされたオブジェクトにとって相談すべき次のコンテキストについての情報を提供する。例えば、図 10.15 の例において、各コンテキストによって使用されているシン

タクス的クラスタリング条件は、そのコンテキストの中で定義された名前のためのコンテキスト結び付けを提供するための手続きとして使用されることもできる。この方式では、構成データが必要ではない、したがってデータ管理のオーバーヘッドはゼロであることに注意されたい。しかし、この方式はテーブル準拠方式よりも柔軟性に乏しい。その理由は、コンテキスト結び付けの変更が必要となっても、各オブジェクトに独立に及び動的にコンテキスト結び付けの変更ができない。コンテキスト結び付けの変更はクラスタリング手続きの変更を必要とする。それゆえ、ほとんどすべての名前付けシステムは、コンテキスト結び付けを実装するために、テーブル準拠方式を使用している。

10.6.4　コンテキスト及び名前リゾルーション機構の分散

名前リゾルーション (name resolution) はあるオブジェクトの名前をそのオブジェクトを**権限名前サーバ (authoritative name server)** に写像する処理である。この処理は基本的に、コンテキストの変換チェーン (resolution chain) の横断 (チェーン中の各コンテキストのコンテキスト結び付け情報を使用しながら) を含んでいる。横断は名前付けされたオブジェクトの権威属性が見つかるまで続けられる。明らかに、ある名前のコンテキストの変換チェーンの横断は、分散システムの中のこれらのコンテキストの位置によって大きく影響を受ける。それゆえ、ある名前付けシステムの名前リゾルーション機構は、その名前付けシステムの名前空間のコンテキストを分散するために使用される方策に依存する。一般的に使用される名前リゾルーション機構の幾つかを以下に述べる。

集中型方式

この方法では、全体の分散システムにおいて 1 つの名前サーバが中央のノードに位置している。この名前サーバはすべてのコンテキストを格納し、名前写像情報を最新に維持する責任をもっている。集中型名前サーバの位置をすべてのノードが知っている。それゆえ、すべてのノードからの名前リゾルーション要求は最初に集中型名前サーバのノードに送信される。名前サーバは、ローカルにコンテキストの完全な変換チェーンをたどることにより名前を変換し、最後に名前付けオブジェクトの属性をクライアントノードに戻す。集中型名前サービスをもつシステムの例は、初期の DARPA の Domain Name System である [Mockapetris 及び Dunlap 1988、Cerf 及び Cain 1983]。

この方式の主な利点は、実装が簡単で容易であるということである。これは

また、どの名前リゾルーション要求もただ2つのメッセージ(要求と応答)がネットワークを通過するのみであるので、効率的でもある。しかし、この方式は幾つかの欠点がある。例えば、大規模システムにおいて集中型名前サーバは重大な隘路ともなりかねないので、規模拡張性に乏しい。さらには、この機構の信頼性もまた非常に低い。ネットワークのリンクの故障により、あるノードと集中型名前サーバのノードとの通信が停止すると、そのノードの名前リゾルーション要求のどれも実行できないという状況になりうる。さらに悪いことはもし集中型名前サーバのノードが故障すると、その場合には、システムのどのノードからのどの名前リゾルーション要求も実行できないことになる。

完全複製方式

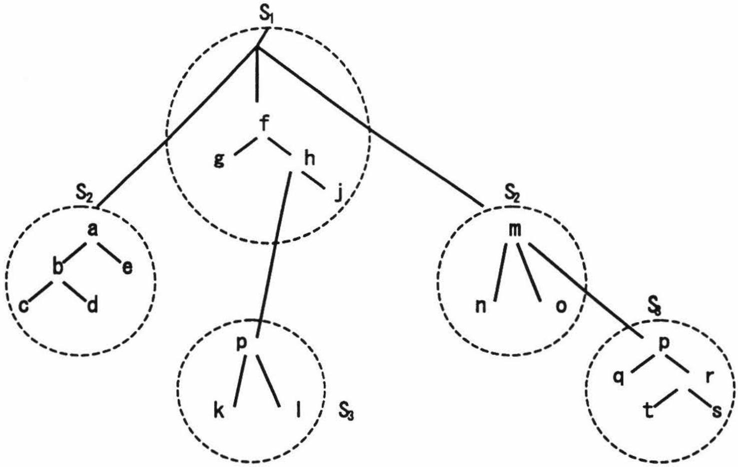
完全複製方式では、分散システムの各ノードに名前サーバが存在し、すべてのコンテキストはすべてのノードに複製されている。すなわち、名前空間データベースは完全に複製され、それゆえ、各名前サーバはすべてのオブジェクト名前についての名前写像情報をもっている。明らかに、どのノードのすべての名前リゾルーション要求もローカル名前サーバによってローカルに処理される。Pup名前サービス［Boggs 1983］及びDDLCN(Distributed Double-Loop Computer Network)システム[Lin 他 1982]はこの方式を使用している。

この方法は、すべての名前リゾルーション要求は、他のどのノードとも通信を行うことなく、ローカルに処理されるので、簡単で効率的である。しかし、この方法はすべてのサーバの中の名前付け情報の一貫性を維持するために大きいオーバーヘッドを必要とするので、大規模名前空間には適していない。

名前空間の物理的構造に基づく分散

これは、階層的ツリー構造の名前空間に対して最もよく使用される方法である。この方法では、名前空間ツリーは幾つかのサブツリーに分割される。それらは、Spite[Welch 及び Ousterhout 1986]では**ドメイン(domain)**と、LOCUS[Walker 他 1983]では**ファイルグループ(file group)**と呼ばれるように、異なるシステムによって、異なる名前で呼ばれる。このサブツリーの概念を以前の用語とは別にするため、ここでは**ゾーン(zone)**という用語を使用する。この方式では、分散システムに幾つかの名前サーバが存在し、各サーバは1つ又はそれより多いゾーンを格納している。図10.18に例を示す。名前空間ツリーの各ノードはあるコンテキストに対応することに注意されたい、それゆえこの方式では、同じゾーンに所属するコンテキストは、同じにグループに集

められる。したがって、この方式では、コンテキストの代わりに、ゾーンが、名前付けされたオブジェクトに関して格納及び複製のための細分できない単位を表す。



(a) ゾーンに分割された階層的名前空間ツリー

名前プレフィックス	ゾーン識別子	
	サーバ識別子	特定ゾーン識別子
/	S_1	12
/a	S_2	19
/f/h/p	S_3	33
/m	S_2	87
/m/p	S_3	61

(b)　(a) の名前ツリーのゾーンに対する名前プレフィックステーブル／キャッシュ

図 **10.18**　名前空間の物理的構造に基づくコンテキストの分散と管理

この方式では、あるオブジェクト名前の変換は基本的に、名前の最後の要素が所属するサブツリー (ゾーン) を管理するサーバへ、名前リゾルーション要求を送信することを含む。例えば、図 10.18 において、オブジェクト名前 /a/b/c、/f/h、/f/g、/f/h/p/k、/m/n、/m/p/r、/m/p/r/s に対する名前リゾルーション要求はサーバ S_2、S_1、S_1、S_3、S_2、S_3、及び S_3 にそれぞれ送信されなければならない。名前の写像をサーバにさせるために、各クライアントは、動的に構築及び更新される名前プレフィックステーブル (**name prefix**

table) (名前プレフィックスキャッシュ(**name prefix cache**) としても知られている) を維持する。名前プレフィックステーブルの各エントリは名前空間ツリーのゾーンの1つに対応する。各エントリはゾーンにおける最上位のコンテキスト (ディレクトリ) の名前 (ゾーンの**名前プレフィックス (name prefix)** と呼ばれる) 及びゾーン識別子 (**zone identifier**) をもっている。ゾーン識別子は2つのフィールドからなる。すなわち、**サーバ識別子 (server identifier)** 及び**特定ゾーン識別子 (specific zone identification)** であり、特定ゾーン識別子はサーバによってゾーンに割り付けられる。それは、同じサーバによって維持されている多くのゾーン間を区別するのに役立つ。例えば、図 10.18 において、サーバ S_3 は2つのゾーンを維持している。それらのルートの両方は同じ名前 p をもっているが、ゾーン識別子は 33 及び 61 と異なる。サーバの識別子はその中に位置情報を持っている。

ある名前が変換されるとき、クライアントは、変換すべきオブジェクト名前と一致する名前プレフィックスを探して自己の名前プレフィックステーブルを探索する。/、/a、/a/b、又は/a/b/c は/a/b/c と名前付けされたオブジェクトに対して、一致する名前プレフィックスであることに注意されたい。もし一致する名前プレフィックスが1つより多く見つかるならば、最大長のものが選択され、クライアントは対応するゾーン識別子情報を使用して、そのゾーンを保持しているサーバに、名前リゾルーションの要求を直接送信する。テーブルエントリの特定ゾーン識別子も、その要求とともに送信される。

サーバが要求を受信すると、サーバは、要求された名前リゾルーションを行うために、要求に伴っている特定ゾーン識別子を使用する。それから、サーバは名前付けされたオブジェクトの残りの要素名前を、選択されたゾーンのコンテキスト (ディレクトリ) で写像する。すべての残りの要素が写像されると、名前リゾルーション操作は完了となり、オブジェクトの属性がサーバによってクライアントに返送される。他方、もし残りの要素名前の一部のみが選択されたゾーンで変換されるならば、さらに名前リゾルーションを続けるために、どれか他のゾーンが使用されなければならない。すなわち、ゾーン切り替えが必要となる。この場合に、さらなる処理のために、サーバは次のゾーンの名前プレフィックスをクライアントに返送する。それからクライアントは、次のゾーンのゾーン識別子を得るために、新しいプレフィックスをブロードキャストする。そのゾーンを格納しているサーバはそのブロードキャストメッセージに応答する。その応答を使用して、新しいゾーンの詳細が名前プレフィックステーブルに中にエントリされ、新しいゾーンのサーバへ要求を送信することにより、以

前と同様に名前リゾルーションが続行する。

名前プレフィックステーブルはブロードキャストプロトコルに伴い、動的に構築及び更新される。最初は、各クライアントは空の名前プレフィックステーブルでスタートする。必要なときのみ、名前プレフィックステーブルのエントリが追加される。すなわち、あるクライアントによってアクセスされないゾーンの名前プレフィックスは、そのクライアントの名前プレフィックステーブルに現れてこない。あるシステムでは、名前空間ツリーのルートディレクトリを含むゾーンの名前プレフィックスはすべてのノードに知られ、その生成の時に名前プレフィックステーブルのエントリとなる。この方式では、クライアントは、名前プレフィックステーブルのどの名前に対する名前プレフィックスを探索する際にも、完全に見つからないことは決してない。

この方式では、あるゾーンの名前プレフィックスは、同じノード又は異なるノードの複数の名前プレフィックステーブルに同時に存在するかもしれない。さらには、あるゾーンのサーバは、可用率、セキュリティ、及び効率を改善するために、動的に変わるかもしれない。それゆえ、この方式の重要な問題は、名前プレフィックステーブルを一貫性を保つためにエントリを更新することである。クライアントがそれらの名前プレフィックステーブルに格納しているゾーンのサーバが変わるときに、すべてのクライアントに知らせる代わりに、名前プレフィックステーブルエントリの一貫性は、使用が古くなったテーブルエントリを検出し、廃却、又は更新することにより一貫性を維持する。すなわち、名前プレフィックステーブルに格納されたサーバ情報は、それらが間違っていると見つけられたときに訂正されるヒントで成り立っている。**ヒント (hint)** とは、正しければ性能を実質的に改善できるが、正しくないとしても意味的に否定的結果を提示しない一式の情報である。最大の性能が得られるように、ヒントはほとんどいつも正しくあるべきである。もちろん、使用に際して自己検証的である情報のみが、この方策に従う。クライアントが古い名前プレフィックステーブルエントリを使用しようとすると、間違ったサーバへ名前要求を送信することを終了する。もしサーバがもはや存在していないと、クライアントは応答を受け取らずタイムアウトとなる。もしサーバが存在しているが、もはや与えられたゾーンを格納していないならば、サーバはその事実をクライアントに応答する。どちらの場合においても、クライアントは、そのテーブルが古くなっていると認識し、エントリを削除し、名前付けされたオブジェクトの最後の要素名前に対応するコンテキストが所属するゾーンに対するサーバ情報を更新するためにブロードキャストを発行する。

この方式は次の利点をもっている。

1. 名前全体ではなく名前プレフィックスでマッチングを行うことは、多数の名前をカバーするのに少数のテーブルエントリですみ、結果として良好な性能となる。例えば、図 10.18 において、1つのテーブルエントリ /a によって、このゾーンにおけるオブジェクトのそれぞれの名前が、ブロードキャスト又はグローバルディレクトリ探索のオーバヘッドなしに直接適切なサーバに進むことが可能となる。プレフィックス名前 /a をもつゾーンは数百又は数千のオブジェクトに対する属性情報を含んでいる。

2. ルートから始まり名前付けされたオブジェクトの最後の要素名前まで1つずつすべてのディレクトリを探索するグローバルディレクトリ探索と比較して、名前プレフィックステーブルは、ディレクトリ探索機構の一部をバイパスする助けとなる。ディレクトリのバイパスは性能及び信頼性において利点をもっている。その理由は1つの名前サーバの故障があっても、クライアントは他のサーバのゾーンで名前リゾルーションを行うことができる。これを説明するため、[Welch 及び Ousterhout 1986] で与えられた例を述べる。2つのゾーン /及び/a をもつシステムにおいて、/のためのサーバは、名前 /a/b/c を変換するために利用可能である必要はない。もしクライアントが既に、その名前プレフィックステーブルの中に /a に対するエントリをもっているならば、クライアントはそのエントリを使用して、そのゾーンに対するサーバと直接通信を行う。そうでなければ、名前 /a/b/c をブロードキャストし、/a に対するサーバはゾーン識別子情報をクライアントに直接応答するであろう。どちらの場合でも、/に対するサーバは含まれていない。このバイパス機構は上位レベルでの競合を減少させる。しかし、上位レベルのディレクトリをバイパスすることはシステムのセキュリティ機構に影響する。これを説明するために、再び [Welch 及び Ousterhout 1986] で与えられた例を述べる。もし名前プレフィックス /a/b/c があり、クライアントが名前 /a/b/c/d を探索し、/a も /a/b も調べないならば、クライアントは /a/b/c をもつサーバと直接通信することになる。これは、ディレクトリ /a 又は /a/b におけるアクセス制御が無視されることを意味する。その結果、すべてのプログラムがすべてのディレクトリへのパスに沿って暗黙の探索許可をもつことになる。もしディレクトリへのアクセスが制限されるべきであるなら、そのレベルのディレクトリ又はそれより下位でのアクセス制御によって制限をされるべきである。

3. 名前プレフィックステーブルの使用中の一貫性検査は、一貫性制御のオーバヘッドを相当に節減する。その理由は格納しているテーブルエントリが不正になるとき、すべてのユーザに知らせる必要がないからである。

コンテキストの構造フリーの分散

もし現在のサービスがデータによって過負荷になる場合、又はワークステーション (ノード) の追加、削除によってシステム規模が変化する場合、名前サービスは再構成されねばならない。再構成はしばしば、あるオブジェクトの権限名前サーバの変更を必要とする。ゾーンベースのコンテキスト分散方法において、オブジェクトの権限名前サーバを変更することは、その名前を変更すること、そのオブジェクトに対応し、現在の権限名前サーバで管理されているゾーンの中のすべてのコンテキストを新しい名前サーバに格納すること、又はそのオブジェクトの新しい権限名前サーバに格納する新しいゾーンを生成することを要求する。新しいゾーンはそのオブジェクトに対応するゾーンのサブゾーンであり、そのオブジェクトの古い権限名前サーバによって管理される。最初の 2 つの解決法は明らかに受け入れられない。第 3 の解決法は、新しいゾーンと同様に古いゾーンにおけるコンテキストのある部分は重複している必要があるかもしれない。その結果、そのオブジェクトの新旧両方の権限名前サーバに、そのようなコンテキストを格納する必要がある。例えば、図 10.18(a) において、もし /a/b/c と名前付けされたオブジェクトに対する権限名前サーバが S_2 から S_4 に変更されるならば、/a、/a/b、及び /a/b/c と名前付けされたコンテキスト、又は /a/b 及び /a/b/c と名前付けされたコンテキストを含む新しいゾーンが S_4 での格納のために生成されねばならないであろう。前者の場合には、2 つのコンテキスト (/a 及び /a/b) が S_2 及び S_4 の両方に格納されるべきであろう。一方後者の場合には、1 つのコンテキスト (/a/b) が重複している必要がある。大規模な名前空間において、このような変更は多くのコンテキストの重複すなわち複製を必要とする。それゆえ、ゾーンベース方式は、あるオブジェクトの権限名前サーバの割り付け、変更において柔軟性は限定される。

名前空間の物理的構造に基づきコンテキストを分散させるこの方法の制限の対策として、幾つかのシステムでは構造フリー名前分散方式を使用している。構造フリー名前分散は、名前空間の部分に対する管理的制御について何ら制限を設けない。それゆえ、この方式では、名前空間のどのコンテキストも、他のコンテキストとは関係なく、どの名前サーバにも自由に格納／移動することが

できる。すなわち、この方式では、各コンテキストはそれ自体のゾーンに所属すると考えられる。この方法に基づく名前サービスでは、あるオブジェクトの権限名前サーバの割り付け／変更において最大の柔軟性をもつことができる。さらには、この方式は、どのようなゾーンが名前空間を形成するかについて名前サーバは同意を必要としないので、名前管理も単純化する。

この方式を使用するシステムにおいて、名前サーバ間にコンテキストを分割及び分散させるために、ある方策が使用される。例えば、Galaxy [Sinha 他 1991a] において、多くのノードの名前サーバの間にコンテキストを分散及び複製することは、名前リゾルーション機構の信頼性を改善するという考えに基づく。この方策の詳細は [Sinha 他 1992] に与えられる。この方式では、名前空間構造は、必要ではないが、名前サーバの間にコンテキストを分割しかつ分散させるために構造を作り上げることもできる。

この方式では、与えられたオブジェクト名前の名前リゾルーション操作は、名前付けされたオブジェクトの権限名前サーバに遭遇するまで、コンテキストごとにコンテキストの完全変換チェーンをたどっていくことにより実行される。例えば、あるオブジェクトのパス名前/a/b/cは、パス名前の次のコンテキスト (/a) の位置を得るために、最初にルートコンテキスト (/) が調べられることから、変換されはじめる。これは、コンテキスト結び付け情報としてルートコンテキストに格納されていることを思い出されたい。得られた位置情報によって、パス名前の次のコンテキスト (/a/b) の位置を得るために、コンテキスト/aが調べられる。最後に、コンテキスト/a/bを調べることにより、名前付けされたオブジェクトの権限名前サーバの情報が得られる。各コンテキストはどのサーバにも自由に配置されることができるので、名前リゾルーション操作はサーバからサーバへと、恐らく、もし関係するサーバが異なるノードに配置されていると1つのノードから別のノードへと移動するかもしれない。このタイプの名前リゾルーション機構において述べるべき2つの重要な疑問がある。

1. どのようにコンテキストオブジェクトが位置特定されるべきか。
2. 名前リゾルーションの間、名前サーバとどのようにやりとりすべきか。

これらの疑問を取り扱う方法を以下に述べる。

名前リゾルーションの間のコンテキストオブジェクトの位置特定:確認された名前はcontext/nameの対からなっていることを思い出されたい。すなわち、あ

る名前は常にあるコンテキストを伴う。それゆえ、あるユーザが変換すべき名前を指定すると、その変換は伴っているコンテキストにおいて開始すべきである。したがって、変換処理における最初のステップはコンテキストの位置特定を行うことである。しかし、コンテキストの位置特定はその名前の変換を含んでいる。すなわち、コンテキスト位置特定操作は最初に名前リゾルーションによって起動される。したがって、ある特別の技法を使用して解決すべき無限再帰がある。この問題の対策として、2 つの通常使用される方法を以下に述べる。

1. **メタコンテキストの使用**: この方法では、**メタコンテキスト (metacontext)** と呼ばれる特別のコンテキストが使用される。これには名前空間におけるすべてのコンテキストオブジェクトに対する、名前及び権威属性の対が含まれている。メタコンテキストの大きさは名前空間におけるコンテキストの数に依存する。もしそれが小さいと、すべての名前サーバノードにそれを格納することができる。しかし、それが大きいと、ある名前サーバノードにのみそれを格納し、他の名前サーバはメタコンテキストを格納するサーバへのポインタのみ格納する。それゆえ、変換すべき名前がコンテキストの中に与えられると、コンテキストの権威属性を得るために、最初にメタコンテキストが調べられる。それから名前がそのコンテキストの権限名前サーバの 1 つに変換のために送られる。

2. **常にルートコンテキストから変換を開始**: この方法は、名前空間が 1 つの全体の階層的名前ツリーとして構造化されている名前付けシステムにおいて使用される。この方法では、ルートコンテキストはすべての名前サーバに複製されている。コンテキストの中に変換すべき名前が与えられると、この名前はそのコンテキストに相対的であると言われる。相対名前 (relative name) は、コンテキスト名前を前に置くことにより、絶対名前に変換されることができる。絶対名前は名前空間のルートコンテキストで始まることを思い出されたい。それゆえ、この方法では、コンテキストの中に変換すべき名前が与えられると、名前は最初に絶対形式に変換され、それからルートコンテキストから変換が開始する。ルートコンテキストはすべての名前サーバノードにおいて利用可能であるので、どの名前の変換もどの名前サーバで開始されることができる。

名前リゾルーションの間の名前サーバとの相互動作. 与えられたパス名前の様々なコンテキストは異なる名前サーバに格納されているかもしれないことを概観した。それゆえ、このような状況におけるパス名前の変換は、パス名前の

1つ又はそれより多くのコンテキストを格納しているすべての名前サーバと相互動作することを含んでいる。名前リゾルーションの間、名前エージェントは以下に述べる方法の1つで、名前サーバと相互動作するかもしれない。

1. **再帰法 (recursive)**: この方法では、与えられた名前の変換を開始するのに必要な最初のコンテキストを格納している名前サーバに、名前エージェントは名前リゾルーション要求を転送する。この後に、与えられたパス名前のコンテキストを格納している名前サーバは、パス名前の最後の要素名前に対応するコンテキストから、名前付けされたオブジェクトの権威属性が抽出されるまで、1つずつ起動される。最後の名前サーバは、前の名前サーバに権威属性を戻す。前の名前サーバはそれ自体の前の名前サーバに権威属性を戻す。以下同様。最後に、名前エージェントから要求を受け取った最初の名前サーバが名前エージェントに権威属性を戻す。図 10.19(a) は再帰相互動作の方法を図示する。

一例として、もし名前/a/b/cが変換を要求されたならば、名前エージェントはそれをルートコンテキスト (/) の名前サーバ (S_1 としよう) に送信し、応答を待つ。それから S_1 はルートコンテキストにおいて要素名前 a を探索し、対応する結び付け情報を抽出し、残りのパス名前 b/c を次のコンテキスト (/a) の名前サーバ (S_2 としよう) に送信し、そして応答を待つ。それから S_2 はコンテキスト /a から要素名前 b に対応する結び付け情報を抽出し、残りのパス名前 c を次のコンテキスト (/a/b) の名前サーバ (S_3 としよう) に送信し、そして応答を待つ。それから S_3 はコンテキスト /a/b から要素名前 c に対応する権威属性を抽出し、S_2 に戻す。それが S_1 に戻され、最後に S_1 が名前エージェントにそれを戻す。

再帰名前リゾルーション機構において、名前エージェントはほとんど処理を行わないが、名前サーバは同時に幾つかの要求の処理に関わるかもしれない。それゆえ、名前サーバは、名前サーバの数に比べて名前エージェントの数が多過ぎる状況では過負荷となるかもしれない。そこで、この機構は、名前エージェントの数と名前サーバの数の比が非常に大きいようなシステムにおける使用には適していない。さらには、名前サーバが応答を待っている間に別の仕事を開始できるためには、名前サーバはマルチプログラムで作成されていなければならない。

2. **繰り返し法 (iterative)**: 図 10.19(b) に示されているように、この方法では、名前サーバは互いに直接呼び出すことはしない。その代わり、名前エー

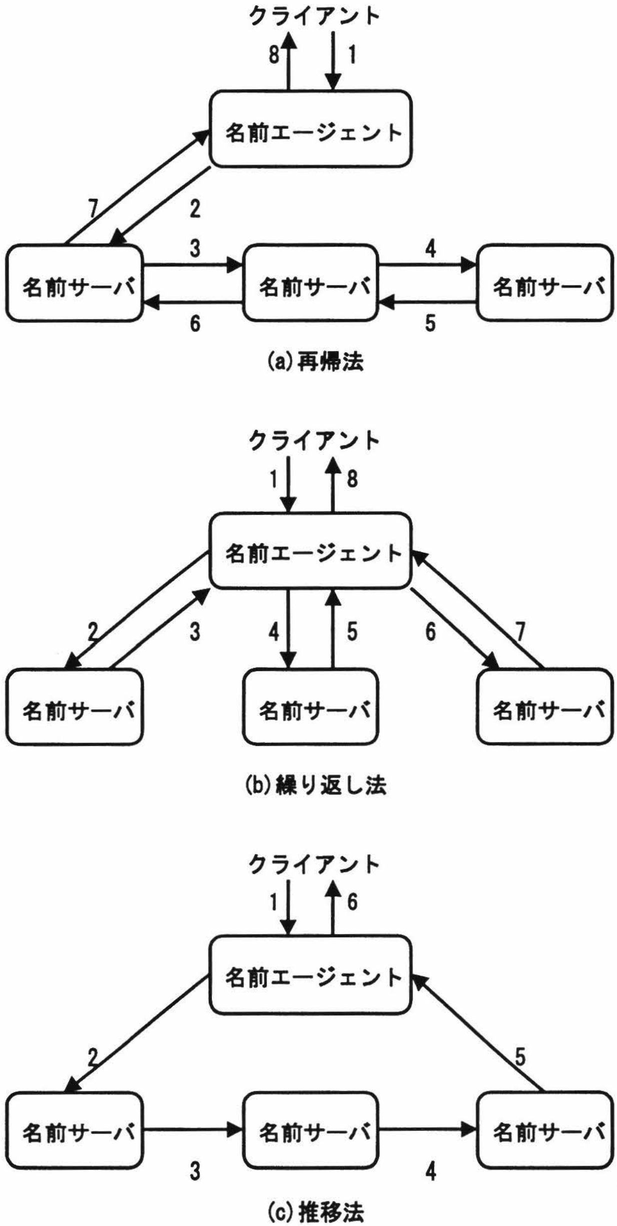


図 10.19　名前リゾルーションのための名前サーバ間の相互動作の方法

ジェントは変換処理についての制御権を保持し、変換処理に関わるサーバのそれぞれを1つずつ呼び出す。再帰処理におけると同様に、名前エージェントは最初に変換すべき名前を、与えられた名前の変換を開始するために必要な最初のコンテキストを格納しているサーバに送信する。サーバは可能な限り多くの名前要素を変換する。もし名前が完全に変換されると、名前付けされたオブジェクトの権威属性がサーバによって名前エージェントに戻される。そうでなければ、サーバは名前エージェントに、名前エージェントが次にコンタクトすべき別の名前サーバの位置情報とともに、変換できなかった名前の部分を戻す。名前リゾルーションを継続するために、名前エージェントは変換できなかった名前の部分を伴って名前リゾルーション要求を次の名前サーバへ送信する。名前エージェントが名前付けされたオブジェクトの権威属性を受信するまで、処理は継続する。

3. **推移法 (transitive)**: この方法において、名前は次のように変換される。名前エージェントが、変換処理を開始するために必要な最初のコンテキストを格納している名前サーバに変換要求を最初に送信する。サーバは可能な限り多くの名前要素を変換する。それから名前サーバは、変換処理を進めるために必要な次のコンテキストを格納している名前サーバに、名前の変換できなかった部分を手渡す。この名前サーバは可能な限り多くの名前要素を変換し、それから次の名前サーバに変換できなかった名前部分を手渡す。処理は継続し、名前付けされたオブジェクトの権威属性に遭遇した名前サーバは、名前エージェントに直接それを戻す。この方法を図 10.19(c) に図示する。

再帰方法におけると同様に、この方法においても、名前エージェントの行う処理はほとんどない。図 10.19 から、推移方法において必要メッセージ数が最も少ないことに注意されたい。しかし、一度他のサーバに変換操作を手渡すと、送信者は確認メッセージを受信しない。それゆえ、この方法は信頼性の高い通信をもつシステムにおいて使用されるべきである。他方、再帰法及び繰り返し法は、RPC(remote procedure call) ベースの通信システムによって効率的にサポートされている。その理由は RPC ベースの通信は「呼び出し応答 (call-response)」モデルを使用しているからである。

10.7　名前キャッシュ

名前リゾルーション操作はかなり複雑な処理であると理解されたであろう。過去の少数の研究者によって行われた測定に基づき、柔軟で階層的な名前空間を提供しているオペレーティングシステムにおいて、名前リゾルーションのオーバヘッドは相当大きいことがわかっている。例えば、Leffler 他 [1984] は UNIX におけるシステム呼び出しオーバヘッドの 40 %がファイル名前リゾルーションにあるとしている。また、Mogul による UNIX システム呼び出し頻度の測定は、名前写像操作 (open、stat、lstat) がファイルシステム呼び出しの 50 %以上となっていることを示している [Mogul 1986]。Sheltzer 他 [1986] もまた、大規模ブロードキャストシステムにおいて、ネットワークトラフィックの実質的な部分は名前に関するものであることを観察している。したがって、名前リゾルーションが必要となるたびに操作を繰り返すよりも、クライアントが名前リゾルーション操作の結果をキャッシュできることが望ましい。

過去において、何人かの研究者［Sheltzer 他 1986、Cheriton and Mann 1989］によって、分散名前キャッシュ(distributed name cache)が、全体システムのオーバヘッドを減少させるのと同様に、名前サービスの性能を改善するのに、適切な解決方法であるかどうかを調べる研究がなされた。これらの研究者によって得られた結論は、単純な分散名前キャッシュが分散システムの性能に実質的な好ましい効果をもつということである。これは、主に名前サービス関連動作の次の特性に起因するものである。

1. **名前探索の高度の局所性:** 「参照の局所性 (locality of reference)」の特性は、データベースのアクセスと同様に、プログラムの実行、ファイルアクセスにおいても観察されてきた。Sheltzer 他［1986］ Cheriton 及び Mann［1989］によってなされた測定は、オブジェクトをアクセスするためのパス名前の使用においても高度の局所性 (high degree of locality) が存在するということを明らかに示している。この局所性により、最近使用された名前付け情報をキャッシュするために使用される適切な大きさの名前キャッシュは良好なヒット率となる。

2. **名前情報データベースの遅い更新:** 名前付けデータはそれほど速く変化しないので、データの不整合は希であることも観察されてきた。ほとんどのユーザの活動は通常、全体の名前情報データベースの小規模で、変化の遅いサブセットに限られる。さらには、ほとんどの名前付けデータは読み取り対更新の

比率が大きい。これはキャッシュデータの一貫性を維持するコストが相当に低いことを意味する。

3. **キャッシュ情報の使用時の一貫性が可能である:** 名前サービス関連動作の都合の良い特性は、古い名前付けデータを使用しようとするとき、何等かの処理が動作できないことを検出でき、使用時点で割り当てることができるという点である。すなわち、名前キャッシュの一貫性は、使用時に古いキャッシュエントリを検出し、削除することによって維持されることができる。使用時一貫性の検査により、名前付けデータの更新が発生するとき、すべての関連するキャッシュエントリを無効にする必要はないし、古いデータがある名前を間違ったオブジェクトに写像することもない。

名前キャッシュの設計に特有の事項が次の節で議論される。

10.7.1 名前キャッシュのタイプ

各エントリに格納された情報のタイプに応じて、**名前キャッシュ**は次のタイプのどれかになるであろう。

1. **ディレクトリキャッシュ(directory cache):** このタイプの名前キャッシュにおいて、各エントリはディレクトリページからなっている。このタイプのキャッシュは通常、繰り返し法の名前リゾルーションを使用しているシステムに使用される。名前リゾルーションの間、クライアントノードにもたらされた最近使用されたすべてのディレクトリページが、暫くの間キャッシュされる。例えば、LOCUS [Sheltzer 他 1986] はこのタイプの名前キャッシュを使用している。LOCUS 設計者がディレクトリページをキャッシュすることにした論点は、ディレクトリの内容をリストにする処理、ワイルドカード引き数を拡張する処理、親ディレクトリをアクセスする処理などの共通の処理はすべてディレクトリページに存在する情報を使用するという点である。しかし、これは、ただ1つの有用なディレクトリのエントリに対しても、ディレクトリの完全なページは、大きい名前キャッシュの領域を必要とし、貴重なキャッシュ空間の大部分をブロックすることになる。

2. **プレフィックスキャッシュ(prefix cache):** このタイプの名前キャッシュ

は、ゾーンベースコンテキスト分散機構を使用している名前付けシステムにおいて使用される。このタイプの名前キャッシュにおいて、各エントリは名前プレフィックス及び対応するゾーン識別子とからなっている。ある名前プレフィックスは、ゾーンベースコンテキスト分散方法におけるゾーンに対応していることを思い出されたい。このタイプの名前キャッシュの使用の詳細は既に 10.6.4 節で述べた。Sprite［Welch 及び Ousterhout 1986］及び V-System［Cheriton 及び Mann 1989］はこのタイプの名前キャッシュを使用している。このタイプの名前キャッシュは構造フリーコンテキスト分散方法に使用するのには適していない。

3. **完全名前キャッシュ(full-name cache)**: このタイプの名前キャッシュにおいて、各エントリは**完全パス名前 (full pathname)** 及び識別子及び権限名前サーバの位置からなっている。それゆえ、名前がローカルのキャッシュに存在するオブジェクトのアクセス要求は、オブジェクトの権限名前サーバに直接送信される。このタイプの名前キャッシュは、構造フリーコンテキスト分散方法を使用する名前付け識別子に主に使用されるが、どのような名前付け機構にも便利に使用される。**プレフィックスキャッシュ(prefix cache)** においては、1 つのエントリは幾つかのオブジェクトに対する写像情報を提供しているが、この**完全名前キャッシュ(full-name cache)** では、各エントリはただ 1 つのオブジェクトのための写像情報を提供している。それゆえ、完全名前キャッシュはプレフィックスキャッシュに比べてサイズが大きい。このタイプの名前キャッシュは Galaxy［Sinha 他 1991b］において使用されている。

10.7.2　名前キャッシュ実装方法

2 つのよく使用される名前キャッシュ実装の方法を次に述べる。

1. プロセスごとのキャッシュ
2. あるノードのすべてのプロセスに 1 つのキャッシュ

両方の方法はそれぞれ利点もあり欠点もある。最初の方法において、各プロセスごとに別々の名前キャッシュが維持される。各キャッシュは対応するプロセスのアドレス空間に維持され、通常サイズは小さい。それゆえ、キャッシュされた情報をアクセスすることは高速であり、名前キャッシュに使用されるオペ

レーティングシステムのメモリ領域を必要としない。しかし、**プロセス指向名前キャッシュ(process-oriented name cache)** はプロセスとともに消滅する。それゆえ、新しいプロセスごとに名前キャッシュを元から生成しなければならない。したがって、もしプロセスが短命であるならば、キャッシュも短命である。そして、新しい、空のキャッシュが生成されるとき発生する初期ミスである**スタートアップミス (start-up miss)** のゆえにプロセス指向キャッシュの実際のヒット率はかなり低いものとなろう。さらには、同じノードの幾つかのキャッシュに同じ名前付け情報が重複している可能性があるので、プロセス指向キャッシュの使用は 1 つのプロセスのみに限定される。スタートアップミスの問題を軽減するために、V-System の実装［Cheriton and Mann 1989］はキャッシュされたデータの寿命を長くするため**キャッシュ継承 (cache inheritance)** を使用している。この方法では、各プロセスは親プロセス (通常は V-System のコマンドインタプリタ又は「シェル (shell)」) から初期キャッシュ内容を継承する。

第 2 の方法では、各ノードのすべてのプロセスに対し各ノードに 1 つの名前キャッシュが維持される。プロセス指向名前キャッシュに比較して、これらのキャッシュはサイズが大きく、オペレーティングシステムのメモリ領域に位置している。キャッシュされた情報のアクセスは、プロセス指向キャッシュのアクセスに比べて遅い。しかし、単一名前キャッシュにキャッシュされた情報は、寿命が長く、キャッシュ置き換え方策が削除することが適切であると判断するときにのみ削除される。それゆえ、単一名前キャッシュ方法にスタートアップミスの問題は関係しない、その結果プロセス指向キャッシュに比べて高い平均ヒット率となる。この方法では、同じノードにキャッシュ情報を重複する可能性もない。

Sprite 及び V-System はプロセス指向名前キャッシュを使用しており、一方 LOCUS 及び Galaxy は単一名前キャッシュ方法を使用している。

10.7.3 マルチキャッシュの一貫性

名前付けデータ (naming data) の更新が発生すると、関連した名前キャッシュエントリは古くなり、無効とされるか又は適切に更新されなければならない。名前キャッシュのマルチキャッシュの一貫性に対して 2 つのよく使用される方法は、**即時無効 (immediate invalidate)** 及び**使用時更新 (on-use update)** である。

即時無効

この方法において、名前付けデータの更新が発生すると、すべての関係する名前キャッシュエントリは即時に無効とされる。これを行うのに 2 つの方法がある。最初の方法では、名前付けデータの更新操作が行われるときは常に、無効にすべきデータを特定するメッセージが、システムの中のすべてのノードにブロードキャストされる。各ノードの名前キャッシュは、その中に更新データが存在しているか調べられる。もし存在していると、対応するキャッシュエントリが無効とされる。この方法は少数のノードをもつシステムの場合に良好に動作するが、多数のノードをもつ大規模ネットワークに対しては使用不可能となる。

ブロードキャストプロトコルの使用を避けるために、第 2 の方法では、名前付けデータの格納ノード (例えば、ディレクトリを格納しているノード) は、各データに対して、データがキャッシュされているノードに対応するノードのリストを保持する。格納ノードが名前付けデータ更新の要求を受信すると、リストに対応したノードのみが、対応するキャッシュエントリを無効にすることを通知される。この方法は、ノード間で共有されている名前付けデータへの更新が低頻度である場合、及びデータが修正されるとき名前付けデータを共有しているノードの数が少ない場合に限って、使用可能である。

使用時更新

これは、名前キャッシュ一貫性を維持するために、よく使用されている方法である。この方法において、名前付けデータ更新が発生するとき、すべての関連するキャッシュエントリを無効にするという動作は何ら行われない。その代わり、クライアントが古いキャッシュデータを使用するとき、使用しようとするデータは不正に指定されているか又は古いとして名前付けシステムから通知される。否定的な応答を受信すると、更新されたデータを得るための必要なステップ (要求のブロードキャスト又はマルチキャスト、又は実装に依存したその他の方法) がとられる。それからそのデータを用いて、古くなったキャッシュエントリを書き換える。

10.8　名前付け及びセキュリティ

幾つかの集中型及び分散オペレーティングシステムの名前付けシステムの重要な仕事は、名前付けオブジェクト及び名前付けデータベースの中の情報の両方への権利のないアクセスを制御することである。システムの様々な資源 (オ

ブジェクト)への権利のないアクセスを制御するために、多くの異なるセキュリティ機構が提案され、オペレーティングシステムにおいて使用されてきた。第11章で、セキュリティ事項を取り扱う。本節はオブジェクトの名前に関係するセキュリティ事項のみを述べる。3つの基本的な名前に関係するアクセス制御機構を以下に述べる。

10.8.1 保護鍵としてのオブジェクト名前

この方法では、あるオブジェクトの名前は**保護鍵 (protection key)** として働く。あるオブジェクトの名前を知っている(すなわちオブジェクトの鍵を持っている)ユーザはこの名前を使用してオブジェクトをアクセスできる。1つのオブジェクトに対して複数の名前を持ちうるシステムでは、あるオブジェクトは複数の鍵を持つこともあることに注意されたい。この場合には、どの鍵でもオブジェクトのアクセスに使用できる。

この方法を使用するシステムにおいて、ユーザがアクセスすることが許可されていないオブジェクトに対してユーザが名前を定義することはシステムによって許可されない。明らかに、もしユーザがオブジェクトに名前付けできないならば、ユーザはオブジェクトを操作できない。この方式はオブジェクト名前は偽造も盗難もなされないという仮定に基づいている。すなわち、あるユーザにとって他のユーザのオブジェクトの名前を得る方法はないし、名前は容易には推察できない。しかし、実際には、オブジェクト名前は一般にニーモニックとなるように選ばれるので、名前はしばしば容易に推察されうる。それゆえ、この方式は信頼性の高いアクセス制御機構を保証しない。この方式の別の制限はアクセス制御のモードを規定する柔軟性を提供しない。すなわち、あるオブジェクトの名前を持っているユーザは、通常そのオブジェクトに対するすべてのタイプのアクセス権を持っている。例えば、あるファイルオブジェクトへの読み取りアクセス権のみをあるユーザに与え、別のユーザには読み取り及び書き込みのアクセス権の両方を与えるということは、この方式のみでは不可能である。

ケーパビリティ

これは上記の方式の制限を克服するための単純な拡張である。図10.20に示すように、**ケーパビリティ(capability)** は保護のための追加の情報を含む特別のタイプのオブジェクト識別子である。これは、その保持者が1つ又はそれよ

り多くのモード (アクセス制御情報部分によって規定される) でそのオブジェクト (オブジェクト識別子部分で特定される) のアクセスを許可する偽造できない切符とみなされるであろう。それゆえ、ケーパビリティは次の特性をもつオブジェクト名前である。

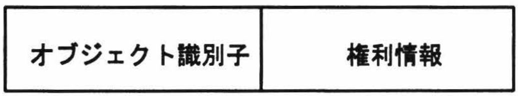


図 **10.20**　ケーパビリティの 2 つの基本部分

1. ケーパビリティはオブジェクトをユニークに特定するシステム指向名前である。

2. オブジェクトを特定することに加えて、特定するオブジェクトの上で行われる操作を定義することにより、参照するオブジェクトを保護するのにも使用される。

3. ケーパビリティを所有するクライアントは、ケーパビリティによって特定されたオブジェクトを、ケーパビリティによって許可されたモードでアクセスできる。

4. 同じオブジェクトに対して、通常幾つかのケーパビリティがある。各ケーパビリティはその保有者に異なるアクセス権を与える。異なる保有者によって保有される同じケーパビリティは、すべての保有者に同じアクセス権を提供する。

5. 与えられたオブジェクトに対するケーパビリティをもつすべてのユーザはこのオブジェクトを共有できる。共有の正確なモードは、同じオブジェクトの各クライアントによって保有されているケーパビリティに依存する。

6. ケーパビリティは偽造できない保護されたオブジェクトであり、オペレーティングシステムによって維持され、ユーザからは間接的にのみアクセスされる。ケーパビリティベースの保護は、ケーパビリティはユーザプロセスから直接

アクセス可能なアドレス空間の中に移動することは許されていないという事実に基づいている。もしすべてのケーパビリティが安全ならば、ケーパビリティが保護しているオブジェクトもまた、権利のないアクセスに対して安全である。

あるプロセスが、あるオブジェクトに、ある操作を行おうとするとき、オブジェクトのケーパビリティを含むメッセージを名前サーバに送信しなければならない。名前サーバは、クライアントによって提出されたケーパビリティが、クライアントによって、あるオブジェクトに要求された操作のタイプを許可しているかどうかを確認する。もし、許可されていないと、「許可否決 (permission denied)」のメッセージがクライアントプロセスに応答される。もし許可されていると、クライアントの要求がオブジェクトの管理者へ転送される。ケーパビリティベースの方法では、ユーザ識別の検査が行われないことに注意されたい。もしこれが必要ならば、ユーザ認証 (user authentication) の何等かの機構が使用されなければならない。第 11 章でこの問題を詳細に取り扱う。

10.8.2　名前リゾルーションパスによる保護

保護はオブジェクトそのもの、又はオブジェクトを特定するのに使用される**名前リゾルーションパス (name resolution path)** に関連して行うことができる。よく使用される方式は名前リゾルーションパスの保護を提供する。

この方法を使用するシステムは通常、**アクセス制御リスト (access control list：ACL)** ベースの保護を使用する。この保護方式はユーザの識別 (identity) に基づきアクセスを制御する (第 11 章に述べる)。ACL に基づく機構は、オブジェクト識別子に加えて、アクセスプリンシパル (accessing principal) を表現する別の**信頼できる識別子 (trusted identifier)**、すなわちアクセス権が関連するエンティティを必要とする。この信頼できる識別子は、偽造も盗難もなされないパスワード、アドレス、又はその他の識別子形式である。ACL はオブジェクトと関連しており、ユーザ名前 (ユーザ識別子) 及びそのオブジェクトの各ユーザに許可されているアドレスのタイプを特定する。あるユーザがあるオブジェクトにアクセスを要求するとき、オペレーティングシステムはそのオブジェクトに関連した ACL を検査する。もしそのユーザがアクセス要求にリストされているなら、そのアクセスは許可される。そうでなければ、保護違反が発生し、ユーザのジョブはオブジェクトのアクセスを否定される。

ACL を名前空間の各コンテキスト (ディレクトリ) に関連させることにより、

名前付けされたオブジェクト及び名前付けデータベースの中の情報へのアクセスを制御することができる。名前サーバがあるディレクトリへのアクセス要求を受信すると、要求されたアクセスに対して、アクセスしているプロセスが権利を持っているかを最初に確認する。この方法では、名前サーバは権利を持っていないクライアントに対して情報を提供しない。それと同時に名前サーバは、名前空間のコンテキストの中に格納された名前付け情報に対する権利のない更新を受け付けない。

あるオブジェクトへのアクセス制御に関して、クライアントはオブジェクトの名前を指定してアクセスする。名前は最初に、名前付けされたオブジェクトの権限名前サーバを見つけるために、コンテキストオブジェクトを使用して変換される。もしオブジェクトの名前に対応するコンテキストに関連している ACL が、クライアントがコンテキスト情報をアクセスすることを許可しないならば、クライアントによって指定された名前は変換されないし、そのクライアントはその名前を使用してそのオブジェクトをアクセスすることは自動的に許可されない。

この方式であるオブジェクトへのアクセスを許可するために、ユーザはオブジェクトのパス名前のディレクトリ及びオブジェクト自体の両方へのアクセスを許可されなければならないことに注意されたい。それゆえ、あるオブジェクト名前の名前リゾルーションパスに関連する保護は、名前付けされたオブジェクトに保護の負荷的な階層を提供する。オブジェクトが複数のパス名前を持ちうるシステム (例えば一般グラフのような) において、あるユーザは、使用するパス名前に応じて異なるアクセス権を持つかもしれないことにも注意されたい。

10.9　事例研究：DCEディレクトリサービス

本章で述べた名前付けの概念及び機構が、分散システムの名前付けシステムを構築するためにどのように使用されるかについての事例研究として、DCE Directory Service(ディレクトリサービス) の概略を以下に述べる。

第 1 章から、DCE システムでは、ユーザ、マシン、及び他の資源はセル (cell) の中にグループ化されることを思い出されたい。それゆえ、DCE ディレクトリサービスはイントラセル及びインターセル名前付けのための次のタイプの要素をもっている。

1. セルディレクトリサービス (**Cell Directory Service：CDS**) :これはセ

ル内の名前付け環境を制御する。各セルは少なくと 1 つの CDS サーバを持っている。

2. **グローバルディレクトリサービス (Global Directory Service：GDS)**:これはセル外の (セル間の) グローバルな名前付け環境を制御する。これは他のセルから位置特定できるように、セル間をリンクする。GDS 実装は X.500 ディレクトリサービスに基づいている。X.500 は CCITT 及び ISO の標準化組織によって定義された名前付けの国際標準である。それゆえ、X.500 スタイルの名前 (後述する) がセルの名前付けに DCE において使用される。

多くの DCE ユーザはインターネットを使用しているので、DCE もまた標準のインターネットの**ドメイン名前システム (Domain Name System：DNS)** をセルの名前付けに使用している。それゆえ、DCE におけるセル名前は DNS 表記法によっても特定される (後述)。

10.9.1　DCE 名前空間及び名前付けコンベンション

DCE はオブジェクトの名前付けに、単一のグローバル名前空間を使用している。すなわち、DCE 名前空間は、シンボル /... で示されるグローバルルートを持つ、1 つの、システム全体の構造である。このルートの下に、各セルを名前付けるのに使用される GDS 名前空間がある。もし DNS 名前が同じ DCE システムにおけるセルを名前付けるためにも使用されているなら、その名前空間もまた、GDS 名前空間の側にいて、DCE グローバルルートの下に現れる。最後に、各セルは、セルルートから始まるそれ自体の内部名前を含む。DCE の名前空間構造の例を図 10.21 に示す。

DCE の各オブジェクトは、各部分がスラッシュで区切られた主に次の部分からなっているユニークな名前をもっている (図 10.22)。

1. **プレフィックス (prefix)**: この部分は、名前が現在のセルにローカルであるのか又は全体の DCE 名前空間にグローバルであるのかを示している。プレフィックス/.:はローカル名前を示すのに使用され、プレフィックス/... はグローバル名前を示すのに使用される。

2. **セル名前 (cell name)**:これは、名前の最初の部分として/... が使用され

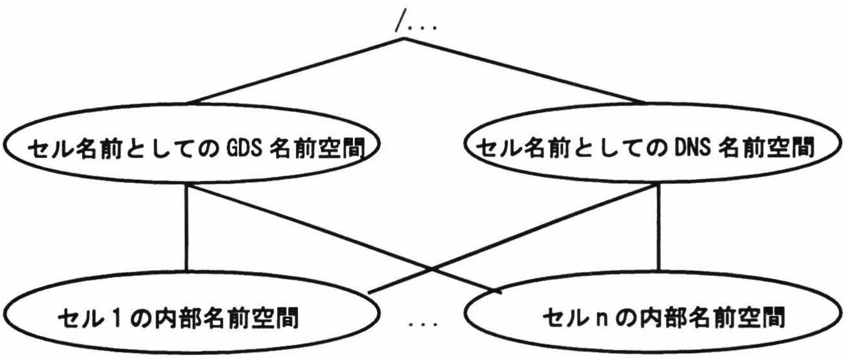


図 10.21　セルの名前付けとして X.500 及び DNS 表記法の両方を使用する n 個のセルをもつ DCE システムの名前空間構造

プレフィックス	セル名前	ローカル名前
/... グローバルを意味する /.:. ローカルを意味する	X.500 表記法又は DNS 表記法	UNIX 類似の階層的名前

ローカルを意味する

図 10.22　DCE 名前付けコンベンション

るときのみ指定されるオプションの部分である。この部分は X.500 の表記法又は DNS 表記法によって指定される。グローバル名前はこの部分を含んでいなければならない。ローカル名前は、この部分を含んでいてはならない。

3. **ローカル名前 (local name)**. この部分はセルの中のオブジェクトをユニークに特定する。UNIX 類似の階層的名前付け方式がローカル名前に使用される。

セルの名前付けのための X.500 及び DNS 表記法を以下に簡単に述べる。

X.500 表記法

OSI の X.500 は、人間、コンピュータ、資源、サービス、その他ユニーク名前が必要な何にでも名前付けを行うための国際標準である。これは、階層的な属性ベースの名前付け方式である (図 10.23)。それゆえ、**X.500 表記法 (X.500 notation)** において、名前はスラッシュで区切られた 1 式の属性によって表現される。各属性は属性タイプ及び 1 つ以上の値をもっている。属性では、タイプと値は等号 (=) によって結ばれている。それゆえ、典型的な X.500 名前は次の形式となる。

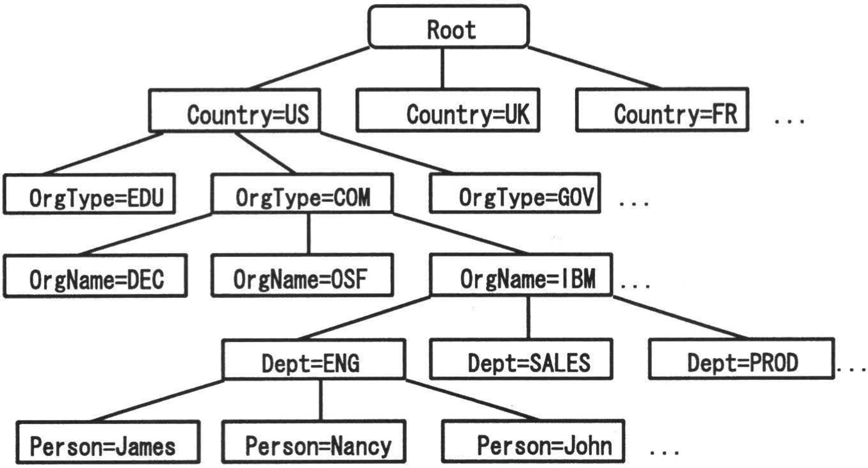


図 **10.23**　X.500 ディレクトリ情報ツリーの一部

/Country=US/OrgType=COM/OrgName=IBM/Dept=ENG/Person=Nancy/

これは、米国の IBM という名前の会社のエンジニアリング部門に所属する Nancy という名前の人をユニークに特定している。X.500 の用語を使用すると、名前の各要素は**相対識別名前 (relative distinguished name : RDN)** と呼ばれ、完全な名前は**識別名前 (distinguished name : DN)** と呼ばれる。

X.500 は、別名 (alias) を定義し使用するための機能を提供し、ファイルシステムにおけるシンボリックリンクに類似する。

X.500 名前ツリーは**ディレクトリ情報ツリー (Directory Information Tree : DIT)** と呼ばれ、ノードに関連するデータを含む全体のディレクトリ構造は**ディレクトリ情報ベース (Directory Information Base : DIB)** と呼ばれる。DIB の 1 つのエントリは OSI の ASN.1 表記法 (抽象構文記法 1 : シンタクス定義の標準的な表記法) によって記述された 1 式の属性からなっている。各属性タイプに対して記述は、タイプ記述、及びタイプの取り得る値すべてについての表現を定義するシンタクス定義を含んでいる。新しい属性を必要に応じて定義できる。

X.500 は、DIB エントリをクラスにグループ化するために、オブジェクト指向情報モデルを使用している。各 DIB エントリは、エントリが参照するオブジェクトのクラス (1 つ又は複数) を決める **ObjectClass** 属性をもっている。例えば、図 10.23 の DIT において、**Country**、**OrgType**、**Dept**、及び **Person** はすべて **ObjectClass** 属性の値の例である。クラスの定義は、与えられたクラスのエントリについて、どの属性が必須で、どれがオプションであるかを決定する。**ObjectClass** 属性は常に必須であり、その値は 1 つ以上のクラスの名前でなければならない。もしあるオブジェクトの **ObjectClass** 属性が 2 つ以上の値をもつならば、このオブジェクトは対応するクラスのそれぞれの必須及びオプションの属性を継承している。

標準である X.500 は、実装問題には対応していない。X.500 についての記述の詳細及び実装方法については、[Rose 1992] を参照されたい。

DNS 表記法

DNS 表記法はインターネット上でホストやその他の資源に名前付けするための標準的な方式である。これは階層的な、ツリー構造で、ドメインに分割された名前空間を使用する。この方式では、名前は、区分記号「.」で区分された 1 つ以上の**ラベル (label)** と呼ばれる文字列からなる。名前の最初と終わりに

は区分記号はない。名前は右の最上位レベルのドメインに書かれる。

インターネット DNS 名前空間は、組織及び地理の両方に従って、分割される。これは世界中を、us(米国)(これはトップレベルドメインでのデフォルトであり、しばしばこのドメインに所属するオブジェクトの確認名前において省略される)、uk(英国)、fr(フランス)、及び jp(日本) などの国名からなるトップレベルのドメインに分割する。異なる国はその国独自の組織のドメインをもち、これが名前空間ツリーの次のレベルのドメインを形成する。例えば、米国では、組織ドメインは、edu(教育機関)、com(民間商用組織)、gov(政府組織)、mil(軍事組織)、net(ネットワーク支援センター)、int(国際組織)、及び org(上記のドメインに含まれていない組織) である。組織ドメインは、stanford.edu、ibm.com、anl.gov、及び ieee.org のようなサブドメインを持っている。これらのサブドメインはまた、cs.stanford.edu のようなサブサブドメインを持っている。

登録管理組織 (registration authority) は、特定レベルでのドメイン内における名前の登録が異なることに責任がある。例えば、米国のスタンフォード教育機関のコンピュータサイエンスの部門を表すドメイン cs.stanford.edu.us はその部門が希望するどのような名前も含ませることができる。しかし、ドメイン stanford.edu.us における名前 cs については、スタンフォード大学の管理組織と合意しなければならない。同様に、ドメイン edu.us における名前 stanford については、ドメイン edu.us を管理している登録管理組織と合意しなければならない。以下同様である。

まとめると、図 10.24 は、DCE におけるあるオブジェクトを参照するために使用し得る 3 つの異なる方法を示している。

10.9.2 DCE におけるイントラセル (セル内) の名前付け

セル内のすべての名前は CDS によって管理される。CDS の機能及び実装を以下に概説する。

CDS ディレクトリ

あるセルの CDS は基本的に、階層的ツリー構造に組織化されたセルのディレクトリを管理する。これらのディレクトリは、セル内のすべてのオブジェクトの名前及び属性を含んでいる。各 **CDS ディレクトリ (CDS directory)** は幾つかのディレクトリエントリを持っている。各ディレクトリエントリは 1 つ

```
/.../ENG.IBM.COM.US/nancy/letters/to/lucy
```

(a)

```
/.../Country=US/OrgType=COM/OrgName=IBM/Dept=ENG/nancy/letters/to/lucy
```

(b)

```
/.:/nancy/letters/to/lucy
```

(c)

図 10.24 DCE ディレクトリサービスにおいて同じオブジェクトを参照するための 3 つの異なる方法。(a) インターネット形式 (DNS 表記法)、及び (b)GDS 形式 (X.500 表記法) により特定されたセル名前でのグローバルオブジェクト名前。(c) セル内のローカルオブジェクト名前。

の名前と 1 式の属性からなっている。例えば、あるプリンタのためのディレクトリエントリは、その名前、場所、タイプ及び性能などの特性を含んでいることであろう。

DCE は、あるオブジェクト名前の名前リゾルーションパスによる関連保護の方法を使用している。それゆえ、各ディレクトリエントリもまたそれと関連する保護情報を持っている。この保護情報は、そのエントリに対し、どのユーザがどのタイプのアクセス権 (読み取り、削除など) を持っているかを規定する。CDS はこの保護情報を管理する。ディレクトリエントリにアクセスするための許可は名前付けされたオブジェクトをアクセスするための許可を意味しない。名前付けされたオブジェクトに対する保護情報は、そのオブジェクトを管理するサーバによって管理される。それゆえ、サーバはどのユーザが、そのオブジェクトの、どのタイプのアクセス権をもっているかを知っている。

名前付け情報の複製

より良い性能と信頼性のために、CDS はその情報の複製、複製の単位はディレクトリをサポートしている。ディレクトリの集まりは**クリアリングハウス (clearinghouse)** を形成する。クリアリングハウスは CDS サーバによって管理される物理的データベースである。各 DCE セルは少なくとも 1 つの CDS サーバで動作しなければならないが、ほとんどは 2 つ以上で複製された重要情報を用いて動作することを選択している。各 CDS サーバは 1 つ以上のクリアリングハウスを維持する。あるディレクトリの各複製は異なるクリアリングハウスに存在する。どの CDS サーバによってもどの名前による探索も始められ

るように、ルートディレクトリはすべてのクリアリングハウスに複製される。これは、新しいディレクトリが生成されるとき、CDSはその親ディレクトリ(すなわち、階層的ツリー構造の名前空間における新しいディレクトリの直接上のディレクトリ)の中にこのディレクトリのためのエントリを自動的に生成する。このエントリは、親ディレクトリ又は子ディレクトリが異なるクリアリングハウスに位置していても子ディレクトリの位置を追跡するために使用される。ルートディレクトリは子ディレクトリのすべてのためのエントリを含む。これらのディレクトリはまた、それら自体の子ディレクトリのためのエントリを含み、以下同様である。それゆえ、ルートディレクトリが与えられると、このパスによる接続は、CDSサーバが、名前空間におけるすべてのディレクトリ(したがってすべてのエントリ)を見つけることを可能とする。

名前付け情報の複製間の一貫性

複製されたディレクトリにおける名前付け情報の一貫性を維持するために、DCEは、ディレクトリ更新操作のために**主コピー (primary-copy) プロトコル**を使用している。すなわち、複製された各ディレクトリのために、1つのコピーが主コピーを指定され、他のすべてのコピーは従コピーである。読み取り操作は複製されたディレクトリのどのコピーを使用して行ってもよい。しかし、すべての更新操作は主コピーに対してのみ行われる。従コピーを更新するのに、次の2つの方法のどちらかが使用される。

1. **更新伝搬: 更新伝搬 (update propagation)** の方法では、ディレクトリの主コピーが更新されると、この変更がすべての従コピーに直ちに送信される。この方法は、一貫性が常時、維持されなければならない名前付け情報の更新に使用される。

2. **スカルキング: スカルキング (skulking)** の方法では、主コピーになされた変更は累積され、1つのメッセージにまとめて送信される。この方法は、クリティカル性の少ない名前付け情報の更新に使用される。

CDS 実装

CDS実装は**クライアントサーバ (client-server)** モデルを使用する。すなわち、**CDS サーバ (CDS server)** デーモンプロセス及び **CDS クライアント (CDS client)** デーモンプロセスが存在する。CDSサーバはサーバマシン上で

動作し、1つ以上のクリアリングハウスを格納及び管理し、ローカルクリアリングハウスにおける名前の生成、修正、又は探索のための要求を処理する。他方、**CDS クラーク (CDS clerk)** と呼ばれる CDS クライアントは、CDS を使用するそれぞれのクライアントマシン上で動作する。CDS クラークはクライアントアプリケーションから要求を受け取り、要求を実行するために 1 つ以上の CDS サーバと協調し、それからクライアントに結果情報を戻す。CDS クラークもまた、名前リゾルーション要求の結果を将来使用のために名前キャッシュに格納する。キャッシュの中の情報がシステムの再立ち上げ又はアプリケーションの再開始にも保存されるように、キャッシュは周期的にディスクに書き込まれる。

CDS クラークが CDS サーバを学習する方法

CDS クラークは、次の方法のどれかで CDS サーバの存在について学習する。

1. **ブロードキャストによる学習**: CDS サーバはそれらの存在を周期的にブロードキャストする。CDS クラークは、これらのブロードキャストメッセージを受信することにより CDS サーバについて学習する。この方法によると、CDS クラークは CDS クラークの LAN と同じ LAN 上に存在するすべての CDS サーバについて学習できる。

2. **名前リゾルーションの間での学習**: 名前リゾルーションの実行中に、もし CDS サーバがそのローカルクリアリングハウスの中の情報で名前を完全に変換できないならば、CDS サーバは、与えられた名前の変換に関して更に多くの情報をもっている別の CDS サーバの位置を要求したクラークに戻す。名前リゾルーション中のこのような CDS サーバからの応答は、CDS クラークが、今まで認識していなかった別の CDS サーバの存在を学習する助けとなる。

3. **管理コマンドによる学習**: DCE 管理者は、CDS クラークのキャッシュにおいて CDS サーバについての情報を生成するために、CDS 制御プログラムを使用することができる。この管理コマンド (management command) による方法は、CDS クラーク及び CDS サーバが異なる LAN 上に存在し、CDS サーバによって送信されたブロードキャストメッセージが異なる LAN 上の CDS クラークによって受信されないときに、通常使用される。

名前リゾルーション

名前リゾルーション操作 (CDS における探索と呼ばれる) は以下に述べるようになされる。説明の順序は図 10.25 のステップに対応しており、DCE において単純な名前リゾルーション操作がどのように行われるかを示している。

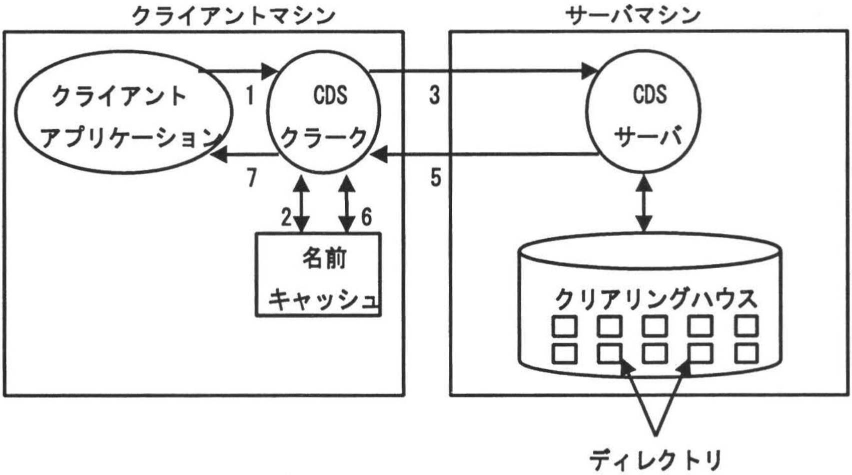


図 10.25　DCE における単純名前リゾルーションのステップ。複数の CDS サーバを含めた複合名前リゾルーションは図 10.19(b) の方法を繰り返し使用する。

1. クライアントアプリケーションが探索要求をローカルの CDS クラークに RPC メッセージで送信する。

2. CDS クラークはそのキャッシュに名前があるか検査する。もしキャッシュに見つけることができると、CDS クラークはクライアントに応答を行い、名前リゾルーション操作は完了する。

3. もし名前がキャッシュに見つからないならば、CDS クラークは名前を知っている CDS サーバに RPC を行う。

4. ローカルクリアリングハウスに存在するディレクトリを使用し、CDS サーバは可能な限り多くの名前要素を変換しようとする。

5. もし名前が完全に変換されると、CDS サーバは名前リゾルーションの結果を CDS クラークに戻す。

6. CDS クラークはこの情報を将来使用のために、そのキャッシュの中に格納する。

7. CDS クラークは最後にクライアントに応答を行い、名前リゾルーション

操作は完了する。

もし上記のステップ 4 においてコンタクトされた CDS サーバによって名前が部分的にしか変換されない場合 (ルートディレクトリはすべてのクリアリングハウスに複製されているので部分的名前リゾルーションは常に可能である)、CDS サーバは、与えられた名前の変換について、より多くの情報を持っている別の CDS サーバの位置を CDS クリアリングハウスに戻す。それから CDS クラークは、名前リゾルーションのために、この新しく学習した CDS サーバと協調動作を行う。このステップは名前が変換されて、CDS クラークが最後にコンタクトした CDS サーバから名前リゾルーションの結果を受信するまで繰り返される。すなわち、名前エージェントと名前サーバとの協調動作の繰り返し方法が名前リゾルーションの間に使用される。CDS クラークは名前リゾルーションの結果及び新しく学習した CDS サーバの情報をキャッシュし、それからクラークに応答を行う。

上記の記述から、CDS は名前リゾルーションを行うのみであることに注意されたい。CDS はオブジェクトアクセス操作を行わない。名前リゾルーションの後に、オブジェクトをアクセスするために、クライアントは名前付けされたオブジェクトを管理しているサーバに対して RPC を行わなければならない。

10.9.3　DCE におけるインターセル (セル間) の名前付け

1 つのセルのあるクライアントが他のセルに所属するオブジェクトをアクセスしようとする。他のセルのオブジェクトを指示する名前を変換するために、CDS クラークは、他のセルの CDS サーバの位置を知る方法を持たなければならない。DCE システムはセル名前として X.500 表記法で格納された GDS 名前空間をもっていることを既に述べた。この名前空間は **GDS サーバ (GDS server)** によって管理される。セル名前としての GDS 名前空間に加えて、DCE システムは、DNS 表記法でのセル名前のための DNS 名前空間も持っている。この名前空間は **DNS サーバ (DNS server)** によって管理される。これらの 2 つの名前空間があるセル名前を、そのセル内の CDS サーバに写像する。

さらに、**グローバルディレクトリエージェント (Global Directory Agent：GDA)** と呼ばれる要素が、他のセルと通信を行うことが必要などのセルにも存在する。これは CDS サーバと同じマシン上か、又は別の独立したマシンのどちらかに存在する。図 10.26 のセル間の名前リゾルーションの例において、

GDA が独立のマシン上に存在すると仮定した。セルは、可用性と信頼性を向上するために 1 つ以上の GDA を持つこともある。あるセルの CDS サーバは、ローカル GDA の位置についての情報を持っている。

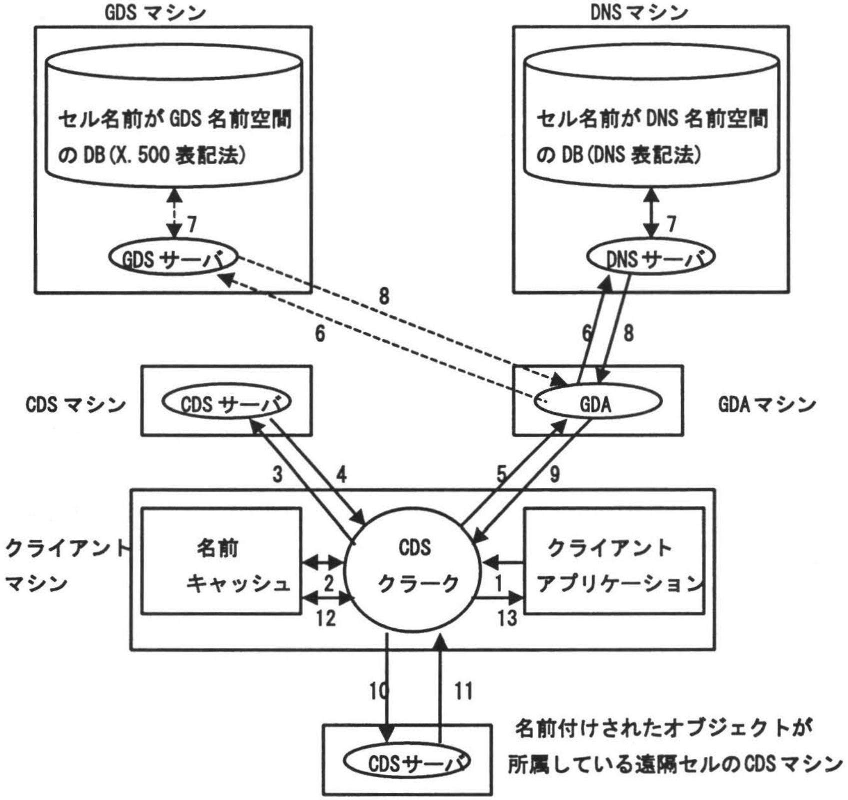


図 10.26 DCE におけるセル間の名前リゾルーション

別のセルのオブジェクトを指示している名前の変換におけるステップを次に示す。次の説明におけるステップは図 10.26 のステップに対応している。そして図 10.26 は DCE において、セル間での名前リゾルーションがどのように行われるかについて示す。

1. あるクライアントアプリケーションがローカル CDS クラークに、RPC メッセージの形で探索要求を送信する。

2. CDS クラークはキャッシュにその名前を検査する。もしキャッシュに見つかれば、CDS クラークはクライアントに応答を行い、名前リゾルーション操作は完了する。

3. もし名前がキャッシュに見つからなければ、CDS クラークは CDS サーバに RPC を行い、GDA の位置を問い合わせる。グローバル名前はセル名前を持たなければならない。そしてローカル名前はセル名前を持ってはならない。それゆえ、クライアントによって与えられた名前を調べることにより、CDS クラークは、それがグローバル名前であり、そのセル名前を変換するために GDA にコンタクトすべきであると知る。

4. CDS サーバは GDA の位置を CDS クラークに戻す。

5. CDS クラークは GDA と RPC を行い、変換すべき名前の中のセル名前を GDA に送信する。

6. GDA はセルの名前にどちらの表記法が使用されているかを検査する。もし X.500 表記法であるなら、GDA は GDS サーバと RPC を行う。他方、もし表記法が DNS 表記法が使用されているなら、GDA は DNS サーバと RPC を行う。

7. GDS 又は DNS サーバはそのデータベースの中のセル名前を探索する。

8. GDS 又は DNS サーバは、名前付けされたセルにおける CDS サーバのアドレスを GDA に戻す。

9. GDA はこの情報を CDS クラークに転送する。

10. CDS クラークはこの情報を使用して、名前探索要求を、名前付けされたオブジェクトが所属するセルの CDS サーバに送信する。CDS サーバは自己のクリアリングハウスのディレクトリを使用して名前を変換する。この例において、コンタクトされた CDS サーバは自己のクリアリングハウスのディレクトリで名前を完全に変換できると仮定している。もしこれがそうでないならば、遠隔セルの他の CDS サーバとの間で協調動作する繰り返しの方法が CDS クラークによって使用される。

11. CDS サーバは、CDS クラークに名前リゾルーションの結果を戻す。

12. CDS クラークは、この情報を将来使用のため、自己のキャッシュに格納する。

13. CDS クラークは最後にクライアントに応答を行い、名前リゾルーション操作は完了する。

10.9.4 DCEディレクトリサービスへのユーザインタフェース

DCE ディレクトリサービスは、名前付け情報に対して異なるアクセス権を持つユーザに対して、次のタイプのユーザインタフェースをサポートする。

1. **ブラウジングインタフェース:** ブラウジングインタフェースは名前付け情報に対して、最低の権利を持つユーザに対するものである。このインタフェースでは、ユーザがセルディレクトリの内容と構造をただ見ることのみが可能である。インタフェースは、OSF/Motif グラフィカルユーザインタフェースに基づくウインドウソフトウェアによりワークステーション上で動作するブラウザツールの形をとる。ブラウザは、ディレクトリの内容を表示するとともに全体のディレクトリ構造も表示する。また特別の種類のディレクトリのみを表示するようにカストマイズすることもできる。各ディレクトリエントリはそれに関連する保護情報を持っていることを思い出されたい。保護情報はどのユーザがどのタイプのアクセス権をエントリに対して持っているかを規定している。ユーザがブラウザツールを使用するとき、ユーザが読み取り許可を持っているディレクトリエントリのみが表示される。他のエントリは表示されない。

2. **XDS アプリケーションプログラミングインタフェース:** DCE ディレクトリサービスに直接呼び出しをできるアプリケーションを書くために、**XDS(X/Open Directory Sever)** アプリケーションプログラミングインタフェースを用いて、ユーザはディレクトリエントリを生成、修正、及び削除することができる。XDS アプリケーションプログラミングインタフェースは、**XOM(X/Open Object Management)** と呼ばれる X.500 への標準インタフェースに準拠している。XDS インタフェースは、ディレクトリオブジェクトを操作するため、及びクライアントとディレクトリサーバとの間のコネクションを確立し初期化するための 13 の呼び出しを持っている。ディレクトリ操作の呼び出しのうちの幾つかは、ディレクトリに新しいエントリを追加する add_entry、ディレクトリからエントリを削除する remove_entry、ディレクトリのすべてのエントリをリストにする list、オブジェクトの属性を読み取るための read、オブジェクトの属性を変更する modify_entry、及びオブジェクトの名前を変更する modify_rdn である。

3. **管理上のインタフェース:** 管理上のインタフェースは名前付け情報に対

して最高の権利をもつユーザのためのものである。このインタフェースでは、管理者がシステム内の名前付け情報の構成及び再構成を行うことが可能である。すなわち、ディレクトリが最も使用されるという考えに基づき、管理者は、異なる CDS サーバのクリアリングハウスの中の様々なディレクトリの分散と複製を決定する。複製されたディレクトリに対して、管理者はまた、これらのディレクトリのそれぞれの中の情報の一貫性制御のために使用される機構 (更新伝搬又はスカルキング) を指定する。さらには、管理者はディレクトリエントリへのアクセスを制御するためにディレクトリエントリに関わるアクセス制御権を変更することもできる。

10.10　まとめ

分散オペレーティングシステムの名前付けシステムは、ユーザ及びプログラムがオブジェクトに文字列の名前を割り当て、その後、それらのオブジェクトを参照するのにこれらの名前を使用することを可能とする。名前付けシステムは、オブジェクトの抽象をユーザに提供し、オブジェクトがどのように、そしてネットワークのどこに実際に位置しているのかを隠蔽する。

分散システムのための良好な名前付けシステムの望ましい特性は、位置透過性、位置独立性、規模拡張性、統一名前付けコンベンション、同じオブジェクトに対する複数のユーザ定義名前の取り決め、グループ名前付け機能、複製透過性、最近傍複製の位置特定、及びすべての複製の位置特定である。

名前は有限のアルファベットから選ばれた 1 式のシンボルからなる文字列である。オペレーティングシステムにおいて広く使用される 2 つの基本的な名前のクラスは、人間指向名前とシステム指向名前である。

名前付けシステムは、オブジェクトへの名前の割り付けに 1 つ以上の名前付けコンベンションを採用している。与えられた名前付けコンベンションに対応する名前の集合は名前空間を形成すると言われる。名前空間はフラットか、又は分割されているかのどちらかである。分割された名前空間の各部分は名前空間のドメインと呼ばれる。1 つのドメインで定義された名前は単純名前と呼ばれる。複合名前は、特別の区分文字で分離された 1 つ以上の単純名前から構成される。

名前サーバは、名前付けされたオブジェクトについての情報を維持し、ユーザがその情報をアクセスできるための機能を提供するプロセスである。分散システムの名前空間は、通常複数の名前サーバによって管理される。各名前サー

バは通常、分散システムにおけるオブジェクトの集合の小さいサブセットについてのみの情報を持っている。オブジェクトについての情報を格納する名前サーバは、そのオブジェクトの権限名前サーバと呼ばれる。

名前サービスの分散及び名前サーバの配置は名前サービスのクライアントにとって透過的であるべきである。この透過性は、名前サーバとクライアントとの間で動作する名前エージェントを通して達成される。

名前は常に、コンテキストと関わっている。あるコンテキストはある名前が適正である環境と考えられる。ある「コンテキスト／名前」の対は、あるオブジェクトをユニークに特定する確認された名前を形成すると言われる。

名前リゾルーションは、あるオブジェクトの名前をそのオブジェクトの権限名前サーバに写像するプロセスである。

使用の便宜上、名前付けシステムは通常、略称、相対名前、総称、及び人間指向名前のためのマルチキャスト名前を提供する。

述語／属性ベースの名前をサポートする名前付けシステムでは、あるオブジェクトを記述する１式の属性によって、名前空間においてオブジェクトのグループの間で、そのオブジェクトはユニークに特定される。

オブジェクトの位置特定は、あるオブジェクトのシステム指向固有識別子をオブジェクトの複製位置に写像するプロセスである。分散システムにおいて通常使用されるオブジェクト位置特定機構は、ブロードキャスト、リングブロードキャスト拡大、識別子の中にオブジェクト位置を符号化、生成者を最初に探索後ブロードキャスト、転送位置ポインタの使用、及びヒントキャッシュ使用しブロードキャストの各方法である。

分散システムにおいて様々なオブジェクトに、システム全体でユニークな人間指向名前を割り付けるための４つの基本的な方法は、オブジェクトのローカル名前をそのホスト名前と組み合わせること、分離名前空間と１つの名前空間に結合すること、明示的な要求によって遠隔名前空間の完全な又は部分的な複製を行うこと、及び１つのグローバル名前空間によることである。

名前空間はクラスタリング条件によって、幾つかのコンテキストに分割される。クラスタリング条件は、ある名前に適用されるとき、与えられた名がクラスタリング条件に指定されたコンテキストの中に存在するか否かによって、真又は偽の値を戻す表現式である。3つの基本的な方法は、アルゴリズム的クラスタリング、シンタクス的クラスタリング、及び属性クラスタリングである。

コンテキスト結び付けは、名前付けされたオブジェクトについての詳細情報をもつ別のコンテキストに格納されているコンテキストとその詳細のコンテキ

ストを格納する名前サーバとを関連付けることである。コンテキスト結び付けを実装するために通常使用される 2 つの方式は、テーブル準拠方式及び手続き準拠法式である。テーブル準拠方式のコンテキストはディレクトリとしても知られている。

ある名前付けシステムの名前リゾルーション機構は、名前付けシステムの名前空間のコンテキストを分散させるために使用される方策に依存する。名前リゾルーションにおいてコンテキストを分散させるために通常使用される方法の幾つかは、集中型の方法、完全複製による方法、名前空間の物理的構造に基づきコンテキストを分散させる方法、及び構造フリーのコンテキスト分散による方法である。

単純な分散名前キャッシュが分散システムの性能向上に実質的な効果があることを研究結果が示している。分散システムにおいて使用される 3 つのタイプの名前キャッシュは、ディレクトリキャッシュ、プレフィックスキャッシュ、及び完全名前キャッシュである。名前キャッシュの実装において通常使用される 2 つの方法は、プロセスごとのキャッシュ、及びノードのすべてのプロセスのためのノードごとに 1 つのキャッシュである。複数キャッシュの一貫性を維持するために通常使用される 2 つの方法は、即時更新及び使用時更新である。

多くの集中型又は分散オペレーティングシステムの名前付けシステムの重要な仕事は、名前付けされたオブジェクト及び名前付けデータベースの両方への権利のないアクセスを規制することである。3 つの基本的な名前付けに関連するアクセス制御機構は、保護鍵としてのオブジェクト名前、ケーパビリティ、及びオブジェクト名前の名前リゾルーションパスと関連した保護である。

10.11　演習問題

10.1 分散オペレーティングシステムの名前付けサブシステムによって行われる主な処理は何か。集中型タイムシェアリングシステムのためのオペレーティングシステムにおいても名前付けサブシステムは必要であるか。もし必要でなければ、その理由をのべよ。もし必要ならば、集中型タイムシェアリングシステムの名前付けサブシステムと分散オペレーティングシステムの名前付けサブシステムとの相違点を述べよ。

10.2「位置透過性」と「位置独立性」の違いを述べよ。どちらがより強力な特性であるか、またその理由は何か。

10.3 分散オペレーティングシステムのオブジェクト位置特定機構は、あるオ

ブジェクトのすべての複製の位置を戻すように設計されている。オブジェクトアクセス要求が最近傍の使用可能な複製の位置に送られるように、得られたオブジェクト位置から、名前エージェントが複製の相対的距離を見つけることができ、このシステムにおいて使用される、機構を提案せよ。

10.4 オペレーティングシステムにおいて使用される人間指向及びシステム指向名前の違いを述べよ。システム指向名前によってのみオブジェクトが特定され、オブジェクトの人間指向名前を持たないシステムを設計することは可能であるか。もし不可能ならば、その理由を述べよ。可能ならば、このようなシステムで起こりうる問題は何かを説明せよ。

10.5 名前空間とは何か。階層的に組織化された名前空間について、階層の数に固定のレベル数を用いる場合と可変のレベル数を用いる場合の相対的な利点及び欠点について論じよ。

10.6 名前空間の次のタイプのそれぞれについて例を挙げよ。

(a) 10 の名前をもつフラット名前空間

(b) それぞれのドメインに 3 つの名前をもつ 4 つのドメインをもつ非階層的名前空間

(c) それぞれのドメインに 3 つの名前をもつ 4 つのドメインをもつ階層的名前空間

(d) 2 つの非入れ子のコンテキストと 4 つの確認名前をもつ名前空間

(d) 2 つのコンテキストと 4 つの確認名前をもつ名前空間。コンテキストの 1 つは他のコンテキストの入れ子となっている。

10.7 20 の名前をもつフラット名前空間の例を挙げよ。この名前空間を、8 より多くの名前を含むコンテキストがないようにコンテキストを分割するために次の方法を使用せよ。

(a) アルゴリズム的クラスタリングの方法

(b) シンタクス的クラスタリングの方法

それぞれの場合のクラスタリング条件を明確にせよ。

10.8 オブジェクト位置特定操作は、もしあるオブジェクトの位置が固有識別子の中に埋め込まれていると、非常に単純化される。この利点にも拘わらず、この方法が望ましくない理由は何か。

10.9 分散オペレーティングシステムにおけるシステム指向名前の生成は、集中型オペレーティングシステムにおける名前の生成とは、どのように異なるか。分散オペレーティングシステムにおけるシステム指向名前の生成に使用される 3 つの異なる方法を挙げよ。

10.10 ある分散システムが、人間指向名前のために単一のグローバル名前空間を使用していると想定する。この名前空間における確認名前は、多くの要素名前をもち長くなりがちである。それゆえに、ユーザがオブジェクトを使用するときごとにオブジェクトの確認名前を指定することは、不便であるかもしれない。この問題を解決するための 2 つの方法を提案せよ。そして、両者の相対的な利点及び欠点を論じよ。

10.11 以下のタイプの分散オペレーティングシステムの名前付けシステムの設計における相違点は何か。

(a) オブジェクト移動及びオブジェクト複製を認めないもの

(b) オブジェクト移動は認めるがオブジェクト複製を認めないもの

(c) オブジェクト複製は認めるがオブジェクト移動を認めないもの

(d) オブジェクト移動及びオブジェクト複製の両者を認めるもの

10.12 分散システムの名前付けシステムのためのオブジェクト位置特定機構を設計する場合を想定する。設計の決定事項に影響を及ぼす要素は何か。

10.13 本章で議論されたオブジェクト位置特定機構のどれ (1 つ以上) が、以下のタイプの分散システムの名前付けシステムにとって適切であるか。

(a) 少数のノードをもつ LAN ベースのシステム

(b) 多数のノードをもつ LAN ベースのシステム

(c) WAN ベースのシステム

(d) オブジェクト移動及びオブジェクト複製を認めないシステム

(e) オブジェクト移動は認めるがオブジェクト複製を認めないシステム

(f) オブジェクト複製は認めるがオブジェクト移動を認めないシステム

(g) オブジェクト移動及びオブジェクト複製の両者を認めるシステム

ある特定の場合に 1 つ以上の機構が適切であるならば、読者はどちらが好ましいか。そしてその理由は何か。

10.14「メタコンテキスト」とは何か。コンテキストに分割される名前空間を使用する名前付けシステムの設計において必ずメタコンテキストが必要であろうか。もしそうであるならば、その理由を説明せよ。もしそうでないならば、どのようなタイプの名前付けシステムが必要であるか。

10.15 ある分散システムに対して、人間指向名前のために、単一のグローバル階層的名前空間、及びコンテキスト結び付け情報を維持するためのディレクトリを使用する名前付けシステムを設計する場合を想定する。設計とプログラミングを単純にするために、ディレクトリの管理を扱う部分を分離し、それをディレクトリサーバの形で実装することを決定したと想定する。この場合に、

ディレクトリサーバとの間で相互動作を行うためのプリミティブの集合を規定せよ。

10.16 名前付けシステムの名前キャッシング方式を設計する場合を想定する。この設計に含まれる主な問題点は何か。その設計で、これらの問題を扱う方法において、名前付けシステムのどのような特性が設計に影響を及ぼすか。

10.17 あるユーザがコマンド open(file_name, mode) を入力すると想定する。ここで file_name は /edu/users/sinha/project1/file1 と特定される。このシステムの名前付けシステムは以下の方法を用いている。

(a) コンテキスト結び付けにテーブル準拠方式を用いる

(b) コンテキストの構造フリーの分散を用いる

(c) 名前リゾルーションに、名前エージェントと名前サーバとの間の相互動作による推移法を用いる

(d) ノード上の各プロセスに対する完全名前キャッシュを用いる

このコマンドを実行するためにオペレーティングシステムの名前付けサブシステムが行わなければならない主な処理のステップを実行順に挙げよ。もし読者が必要ならば、他の仮定を設けてもよいが、その場合は仮定を明確にせよ。

10.18 使用時一貫性制御機構を説明せよ。この機構においてどの点が、キャッシュ一貫性のために通常使用されている他の機構に比べ、より良好であるか。この機構はすべてのタイプのキャッシュデータ一貫性制御に使用できるか。読者の答えの理由を述べよ。

10.19 ある分散システムにおいて、オブジェクトのシステム指向名前はオブジェクトへのアクセスを制御するために使用される。すなわち、このシステムにおいて、あるオブジェクトのシステム指向名前を所有する主体 (ユーザ) は、このオブジェクトのすべてのモードでのアクセスを許される。このシステムの設計における重要な問題は、システム指向名前は推察が困難でなければならないという点である。この問題を扱うために使用される、何らかの方法を提案せよ。

10.20 ある分散システムにおいて、オブジェクトのシステム指向名前は、オブジェクトのアクセスを制御するためのケーパビリティとして使用される。すなわち、システム指向名前は、同じオブジェクトの異なるユーザへの異なるアクセス許可を与えるケーパビリティとしても使用できる。このシステム指向名前の生成に使用される方法を考案せよ。

参考文献

[Birrell et al. 1982] Birrell, A. D., Levin, R., Needham, R., M., and Schroeder, M. D., "Grapevine: An Exercise in Distributed Computing,"Communications of the ACM, Vol. 25, No. 4, pp. 260-274 (1982).

[Boggs 1983] Boggs, D. R., "Internet Broadcasting," Ph.D. Dissertation, Stanford University, Technical Report No. CSL-83-3, Palo Alto Research Center(1983).

[Brownbridge et al. 1982] Brownbridge, D. R., Marshall, L. F., and Randell, B., "The Newcastle Connection or UNIXes of the World Unite!" Software Practice and Experience, Vol. 12, No. 12, pp. 1147-1162 (1982).

[Cerf and Cain 1983] Cerf, V. G., and Cain, E., "The DoD Internet Architecture Model," Computer Networks, Vol. 7 ,No. 5, pp. 307-318 (1983).

[Cheriton and Mann 1989] Cheriton, D. R., and Mann, T. P., "Decentralizing a Global Naming Service for Improved Performance and Fault Tolerance," ACM Transactions on Computer Systems, Vol. 7, No. 2, pp. 147-183 (1989).

[Comer and Murtagh 1986] Comer, D., and Murtagh, T. P., "The Tilde File Namming Sscheme," In: Proceedings of the 6th International Conference on Distributed Computing Systems, IEEE Press, New York, NY, pp. 509-514 (1986).

[Comer and Peterson 1986] Comer, D., and Peterson, L. L., "A Model of Name Resolution in Distributed Systems," In: Proceedings of the 6th International Conference on Distributed Computing Systems, IEEE Press, New York, NY, pp. 523-530 (1986).

[Coulouris et al. 1994] Coulouris,k G. F., Dollimore, J., and Kindberg, T., Distributed Systems Concepts and Design, 2nd ed., Addison-Wesley, Reading, MA (1994).

[Goscinski 1991] Goscinski, A., Distributed Operating Systems, The Logical Design, Addison-Wesley, Reading, MA (1991).

[Kille 1992] Kille, S., Implementing X.400 and X. 500: The PP and QUIPU Systems, Artech House, Norwood, MA (1992).

[Lampson 1986] Lampsoon, B. W., "Designing a Global Name Service," In: Proceedings of the 5th Annual Symposium on Principles of Distributed Computing, Calgary, Canada, Association for Computing Machinery, New York, NY, pp. 1-10 (August 1986).

[Lantz et al. 1985] Lantz, K., Edighoffer, J., and Hitson, B., "Towards a Universal Directory Service," In: Proceedings of the 4th Annual Symposium on Principles of Distributed Computing, Minaki, Canada, Association for Computing Machinery, New York, NY, pp. 250-260 (August 1985).

[Leach et al. 1982] Leach, P. J., Stump, B. L., Hamilton, J. A., and Levine, P. H., "UIDs as Internal Names in a Distributed File System," In: Proceedings of the 1st Annual Symposium on Principles of Distributed Computing, Ontario, Canada, Association for Computing Machinery, New York, NY, pp. 34-41 (August 1982).

[Leach et al. 1983] Leach, P. J., Levine, P. H., Douros, B. P., Hamilton, J. A., Nelson, D. L., and Stumpf, B. L., "The Architecture of an Integrated Local Network," IEEE Journal on Selected Areas in Communication, Vol. SAC-1, No. 5, pp. 842-857 (1983).

[Leffler et al. 1984] Leffler, S., Karels, M., and McKusick, M., "Measuring and Improbing the Performance of 4.2BSD," In: Proceedings of the 1984 USENIX Summer Conference, USENIX Association, Berkeley, CA, pp. 237-252 (1984).

[Levy and Silberschatz 1990] Levy, E., and Silberschatz, A., "Distributed File Systems: Concepts and Examples," ACM Computing Surveys, Vol. 22, No. 4, pp. 321-374 (1990).

[Lin et al. 1982] Lin, M. T., Tsoy, D. P., and Lian, R. C., "Design of a Network Operating System for the Distributed Double-Loop Computer Network (DDLCN)," Local Computer Networks, North-Holland Co., IFIP (1982).

[Lockhart, Jr. 1994] Lockhart, Jr., H. W., OSF DCE: Guide to Developing Distributed Applications, IEEE Computer Society Press, Los Alamitos, CA (1994).

[Mockapetris 1984] mockapetris, P. V., "The Domain Name System," In: Proceedings IFIP 6.5, International Symposium on Computer Messaging, Nottingham, England (May 1984).

[Mockapetris and Dunlap 1988] mockapetris, P. V., and Dunlap, K. J., "Development of the Domain Name System," In: Proceedings of the SIGCOMM'88 Symposium on Communications Architectures and Protocols, Stanford, CA, Association for Computing Machinery, New York, NY, pp. 123-133 (1988).

[Mogul 1986] Mogul, J. C., "Representing Information About Files," Ph.D. Dissertation, Stanford University, Computer Science Technical Report, STAN-CS-86-1103 (March 1986).

[Mullender and Tanenbaum 1986] Mullendr, S. J., and tanenbaum, A. S., "The Design of a Capability Based Distributed Operating System," The Computer Journal, Vol. 29, No. 4, pp. 289-300 (1986).

[Mullender et al. 1990] Mullender, S. J., Van Rossum, G., Tanenbaum, A. S., Van Renesse, R., and Van Staverene, H., "Amoeba: A Distributed Operating System for the 1990s," IEEE Computer, Vol.23, No.5, pp. 44-53(1990).

[Needham 1993] Needham, R. M., "Names," In: S. Mullender (Ed.), Distributed Systems, 2nd ed., Association for Computing Machinery, New York, NY, pp.315-327 (1993).

[Oppen and Dalal 1983] Oppen, D. C., and Dalal, Y. K., "The Clearinghouse: A Decentralized Agent for Locationg Named Objects in a Distributed Environment," ACM Transactions on Office Information Systems, Vol. 1, No. 3, pp., 230-253 (1983).

[Peterson 1988] Peterson, L., "The Profile Naming Service," ACM Transactions on Computer Systems, Vol. 6, No. 4, pp. 341-364 (1988).

[Rashid 1987] Rashid, R. R., "Mach: A New Foundation for Multiprocessor Systems Development," In: Proceedings of COMPCON'87—Digest of Papers, IEEE Press, New York, NY, pp. 192-193 (1987).

[Rose 1992] Rose, M. T., The Little Black Book: Mail Bonding with OSI Directory Services, Prentice-Hall, Englewood Cliffs, NJ (1992).

[Rosenberry et al. 1992] Rosenberry, W., Kenney, D., and Fisher, G., OSF DISTRIBUTED COMPUTING ENVIRONMENT, Understanding DCE, O'Reilly & Associates, Sebastopol, CA (1992).

[Rozier et al. 1988] Rozier, M., Abrossimov, V., Armand, F., Boule, I., Gien, M., Guillemont, M., Herrmann, F., Kaiser, C., Leonard, P., Langlois, S., and Neuhauser, W., "Chorus Distributed Operating System," Computing Systems, Vol. 1, pp. 305-379 (1988).

[Saltzer 1982] Saltzer, J. H., "On the Naming and Binding of Network Destinations," In: Proceedings of IFIP/TC6 International Symposium on Local Computer Networks, Florence, Italy, pp. 311-317 (April 1982).

[Sandberg et al. 1985] Sandberg, R., Goldberg, D., Kleinman, S., Walsh, D., and lyhon, B., "Design and Implementation of the SUN Network File System," In: proceedings of the USENIX Conference, Portland, OR, USENIX Association, Berkeley, CA, pp. 119-130 (1985).

[Schantz et al. 1986] Schantz, R. E., Thomas, R. H., and Bono, G., "The Architecture of the Cronus Distributed Operating System," In: Proceedings of the 6th International Conference on Distributed Computing Systems, IEEE Press, New York, NY, pp. 250-259 (1986).

[Schickler 1982] Schickler, P., "Naming and Addressing in a Computer Based Mail Environment," IEEE Transactions on Communications, Vol.COM-30, No.1 (1982).

[Schroeder et al. 1984] Schroeder, R., Birrell, A. D., and Needham, R. M., "Experience with Grapevine: The Growth of a Distributed System," ACM Transactions on Computer Systems, Vol.2, No. 1, pp. 3-23 (1984).

[Schwartz et al. 1987] Schwartz, M., Zahorjan, J., and Notkin, D., "A Name Service for Evolving Heterogeneous Systems," In: Proceedings of the 11th ACM Symposium on Operating Systems principles, Association for Computing Machinery, New York, NY, pp. 52-62 (November 1987).

[Sheltzer et al. 1986] Sheltzer, A. B., Lindell, R., and Popek, G. J., "Name Service Locality and Cache Design in a Distributed Operating System," In: Proceedings of the 6th International Conference on Distributed Computing Systems, IEEE Press, New York, NY, pp. 515-522 (1986).

[Silberschatz and Galvin 1994] Silberschatz, A., and Galvin, P. B., Operating Systems, Concepts, 4th ed., Addison-Wesley, Reaading, MA (1994).

[Sinha et al. 1991a] Sinha, P. K., Maekawa, M., Shimizu, K., Jia, X., Ashihara, H., Utsunomiya, N., Park, K. S., and nakano, H., "The Galaxy Distributed Operating System," IEEE Computer, Vol. 24, No. 8, pp.34-41(1991).

[Sinha et al. 1991b] Sinha, P. K., Shimizu, K., Utsunomiya, N., Nakano, H., and Maekawa, M., "Network Transparent Object Naming and Locating in the Galaxy Distributed Operating System," Journal of Information Processing, Vol. 14, No. 3, pp. 310-324 (1991).

[Sinha et al. 1992] Sinha, P. K., Maekawa, M., and Shimizu, K., "Improving the Reliability of Name Resolution Mechanism in Distributed

Operating Systems," In: Proceedings of the 12th International Conference on Distributed Computing Systems, Yokohama, Japan, pp. 589-596 (June 1992).

[Su and Postel 1982] Su, Z., and Postel, J., "The Domain Naming Convention for Internet User Applications," Network Information Center, SRI International, RFC 819 (August 1982).

[Terry 1984] Terry, D. B., "An Analysis of Naming Conventions for Distributed Computer Systems," In: Proceedings of the 6th International Conference on Distributed Computing Systems, IEEE Press, New York, NY, pp. 502-508 (1984).

[Terry 1986] Terry, D. B., "Structure-Free Name Management for Evolving Distributed Environments," In: Proceedings of the ACM SIGCOMM'84, Montreal, Quebec, Association for Computing Machinery, New York, NY, pp. 218-224 (June 1986).

[Tichy and Ruan 1984] Tichy, W. F., and Ruan, Z., "Towards a Distributed File System," In: Proceedings of the Summer USENIX Conference, USENIX Association, Berkeley, CA, pp. 87-97 (June 1984).

[Tripathi et al. 1987] Tripathi, A., Ghonami, A., and Schmitz, T., "Object Management in the NEXUS Distributed Operating System," In: Proceedings of IEEE COMPCON'87, IEEE Press, New York, NY, pp.50-53 (February 1987).

[Walker et al. 1983] Walker, B., Popek, G., English, R., Kline, C., and Thiel, G., "The LOCUS Distributed Operating System" In: Proceedings of the 9th ACM SIGOPS Symposium on Operating Systems Principles, Association for Computing Machinery, New York, NY, pp. 49-70 (1983).

[Watson 1981] Watson, R., "Identifiers (Naming) in Distributed Systems," In: B. W. Lampson, M. Paul, and H., Siegert (Eds.), Lecture Notes in Computing Science: Distributed Systems— Architecture and Implementation, Springer-Verlag, New York, NY, pp. 191-210 (1981).

[Welch and Ousterhoout 1986] Welch, B., and Ousterhout, J. K., "Prefix Tables: A Simple Mechanism for Locating Files in a Distributed System," In: Proceedings of the 6th International Conference on Distributed Computing Systems, IEEE Computer Society Press, Los Alamitos, CA, pp. 184-189 (1986).

[Wiebe 1986] Wiebe, D., "A Distributed Repository for Immutable Persistent Objects," In: Proceedings of OOPSLA'86, Association for Computing Machinery, New York, NY, pp. 453-465 (1986).

インターネット参考文献

名前付けのみを扱った文献は見つけることができなかったが、本章に関係するトピックスの参考文献は以下のところで見つけることができる。

ftp:ftp.cs.umanitoba.ca/pub/bibliographies/Os/IMMD_IV.html
ftp:ftp.cs.umanitoba.ca/pub/bibliographies/Misc/misc.1.html
ftp:ftp.cs.umanitoba.ca/pub/bibliographies/Distributed/rfc.html

第11章

セキュリティ

11.1 はじめに

コンピュータシステムは大量のデータを格納し、その中にはユーザにとって極めて重要なものがある。ユーザは、コンピュータシステムの資源や情報が破壊や許可されていないアクセスから保護されている場合に限って、そのコンピュータシステムを信用することができる。明らかに、想定される動作環境に依存して、コンピュータシステム毎に、セキュリティに関する要求は異なる。例えば、軍事環境で動作することを目的としたシステムに対するセキュリティ要求と教育環境で動作することを目的としたシステムのそれとは異なる。あるコンピュータシステムのセキュリティ目標はそのシステムのセキュリティ方針によって決まり、それらの目標を達成するための方法はセキュリティ機構と呼ばれる。システムのセキュリティを設計する段階で、セキュリティ方針とセキュリティ機構を区別することが重要である。これは、セキュリティ方針は利用可能な技術からは独立に決定することができるが、セキュリティ機構はその時点で利用可能な技術に影響を受けるからである。ある時点で選択されたセキュリティ機構では、所望のセキュリティ方針を実現することが困難な場合もある。しかし、所望のセキュリティ方針を実現するために、後になって新しいセキュリティ機構を追加することは可能である。

動作環境とは無関係に、コンピュータセキュリティの幾つかの共通目標は以下の通りである [Mullender 1985]。

1. **安全性** (security):システム内の情報が認許されたユーザによってのみアク

セス可能でなければならない。

2. **プライバシ** (privacy):情報の悪用を防がなければならない。即ち、あるユーザに与えられた一片の情報は、与えられた目的のみに利用されるべきである。

3. **信憑性** (authenticity):ユーザがあるデータを受け取った時、そのユーザはそのデータの信頼性を検証できなければならない。即ち、そのデータが確かに期待している送り手から送られたものであり、他の情報源からのものではないことを検証できなければならない。

4. **完全性** (integrity):システム内の情報は偶然の破壊や認許されていないユーザによる故意の改悪から保護されなければならない。

コンピュータセキュリティ全体は外部セキュリティと内部セキュリティからなる。**外部セキュリティ**(external security) とは外部要因に対してコンピュータシステムを安全に保つことである。ここで、外部要因とは、火事、洪水、地震、ディスクやテープの盗難、アクセス権を持つ人による情報の漏洩などである。外部セキュリティに関して通常用いられる方策は以下の通りである。即ち、格納されている情報の十分なバックアップコピーを、元の情報からは離れた場所に維持すること、コンピュータセンターへ認許された人物だけが入れるように保安員を置くこと、信頼できる従業員やユーザにのみに重要な情報のアクセス権を与えることなどである。

一方、**内部セキュリティ**(internal security) は主に以下の 2 つの側面を扱う。

1. **ユーザ認証** (user authentication):ユーザがコンピュータ設備への物理的なアクセスを許可された場合、そのユーザが実際に設備を使い出す前に、システムはそのユーザの身元を確認しなければならない。

2. **アクセス制御** (access control):コンピュータシステムは多くの資源と各種の情報を含んでいる。明らかに、すべての資源や情報がすべてのユーザにとって意味があるとは限らない。したがって、ユーザが認証過程を経てコンピュータ設備の使用を許可されても、そのユーザのアクセスが許可されていない資源や情報へのアクセスを禁止する方法が必要である。実際のところ、安全なシス

テムでは、任意の時点で対象物 (人間又はプログラム) は、仕事を完了するためにその時点で必要な資源に対してのみアクセスが許可されるべきである。この要求は通常、**必要性原則** (need-to-know principle) 又は**最小特権原則** (principle of least privilege) と呼ばれている。

コンピュータシステムにおけるセキュリティの必要性は、本質的にそのシステムの環境、用途及び実装に関連している。したがって、上述の 2 つの側面に加えて、分散システムにおける内部セキュリティは通信セキュリティと呼ばれる第三の側面を持つ。

3. **通信セキュリティ**(communication security):分散システムでは、コンピュータを接続する通信チャネルは、常に攻撃者に晒されている。攻撃者は通信を傍受し、改ざん又は混乱させることによってシステムのセキュリティを破ろうとする。無線ネットワークは侵入者による傍受にことさら弱い。これは、誰でもスキャナーを持っていれば、検出されることなく無線信号を受信することができるからである。通信セキュリティは、あるコンピュータから他のコンピュータへ通信チャネルを通して情報が転送される途上で、その情報に対して行われる認許されていない干渉に対する保護を提供する。通信セキュリティの他の 2 つの側面は、**通信実体の信憑性** (authenticity of communication entities) と**メッセージの完全性** (integrity of messages) である。即ち、メッセージの送信者はそのメッセージが意図した受信者によって受信されたことを知りたいし、受信者はそのメッセージが本当の送信者から送られてきたことを知りたい。明らかに、送信者と受信者は共に、転送途上でメッセージの内容が変更されていないことを保証してほしい。

外部セキュリティと内部セキュリティの双方を提供することは、集中型システムよりも分散システムの方が難しい。これは、分散システムでは、単一の制御点というものが存在しないことと、データ通信において不安全なネットワークを使用するからである。外部セキュリティは内部セキュリティと同等に重要なものであるが、コンピュータセキュリティに関してオペレーティングシステムのレベルで用いられる方針と機構は内部セキュリティの側面だけを扱っている。したがって、本章では、分散システムにおける異なったタイプの内部セキュリティを提供するために通常用いられている機構のみを扱う。

11.2 コンピュータシステムに対する潜在的な攻撃

適当なコンピュータセキュリティを用意する第一段階は、コンピュータシステムに対する潜在的な脅威／攻撃を特定することである。**侵入者** (intruder) 又は**攻撃者** (attacker) という用語は、通常、コンピュータシステムの資源やデータに対する認許されていないアクセスを獲得しようとする人又はプログラムを意味するのに用いられる。侵入者は、多くの方法でコンピュータに対する脅威となり、それらは大きく分けて受動的攻撃と能動的攻撃の 2 つに分類される。**受動的攻撃** (passive attack) は、脅威に晒されているシステムには何ら害を及ぼさないが、**能動的攻撃** (active attack) は害を及ぼす。したがって、受動的攻撃は本来システムによって検知されるものではなく、予防的的手段によってのみ扱うことができるものである。一方、能動的攻撃は、予防、検出及び回復の技術によって対処されるものである。以下では、これらの攻撃に関する説明と関連する問題について述べる。

11.2.1 受動的攻撃

受動的攻撃では、侵入者はシステムの通常の動作に干渉することなく、コンピュータシステムから許可されていない情報を何とか盗もうとする。よく用いられる受動的攻撃の方法は以下の通りである。

1. ブラウジング (browsing):この方法では、侵入者はデータを変更することなく、格納されたファイル、ネットワーク上を流れるメッセージパケット、他のプロセスのメモリなどを読もうと試みる。格納されたファイルや他のプロセスのメモリ内容に対する許可されていない読み出しを防ぐためには、アクセス制御機構が用いられ、ネットワークリンクを転送されるメッセージの盗聴に対してはメッセージの暗号化が用いられる。

2. 漏洩 (leaking):この方法では、侵入者は情報を漏らしてくれる共犯者 (盗むべき情報に対するアクセス権を有する合法的なユーザ) を用いる。漏洩の防止は、解決困難な問題であり、共犯者と侵入者の間のあらゆるタイプの通信を防止する必要がある。潜在的な共犯者が任意の情報を外部に漏らすことが不可能であることを保証する問題は、**監禁問題** (confinement problem)[Lampson 1973] と呼ばれる。後で述べるように、理論的に通信を全く行えないようなプ

ロセス間の情報漏洩は比較的容易である。したがって、監禁問題は一般には解決不可能である。

3. **推定** (inferencing):この方法では、侵入者はシステムのデータやシステムによって行われる活動をよく観察し、解析することによって、何らかの推定を行おうとする。例えば、許可されていないアクセスから守るために情報が暗号化されている場合、侵入者は暗号化されたデータの断片を解析することによって暗号化鍵を導こうとするかもしれない。導かれた鍵はシステムから情報を盗むために用いることができるため、価値あるものとなり、他の侵入者に売られることもあり得る。推定に関するその他の例としては、分散システムにおけるトラフィック解析がある。この場合には、侵入者はいつ、どこでプロセス間のメッセージの流れが発生するかを観察する。そして、各種通信パターン間のメッセージ交換頻度を解析することによって、侵入者は何らかの推定を引き出そうとする。例えば、ビジネス環境では、異なる組織間で発生する交渉に関する有益なヒントを与えることもあり得る。

4. **偽り** (masquerading):この方法では、侵入者は許可されていないデータや資源へのアクセスを獲得するために、許可されたユーザやプログラムを偽る。例えば、多くのシステムでは、あるユーザによって書かれたプログラムを、他のユーザによって使用可能とするための機構を持っている。これらのプログラムは、実行ユーザのアクセス権を不正に使用し、情報を漏らすことができる。例えば侵入者は、完全にエディタとして動作するが、編集したファイルのコピーを侵入者がアクセス可能な特別の領域に生成するようなエディタプログラムを書くことができる。このエディタプログラムはコンパイルされ、侵入者が関心を持っているファイルを所有するユーザの bin ディレクトリに読み込まれる。それ以降、侵入者はそのユーザが編集したすべてのファイルのコピーを得ることができる。そのエディタプログラムはユーザの編集作業を通常と全く同じように行うので、そのユーザは盗みが行われていることに全く気付かない。

このような方法でコンピュータセキュリティに入り込むことは**トロイの馬攻撃** (Trojan horse attack) として知られている。即ち、**トロイの馬プログラム** (Trojan horse program) とは、通常の機能に加えて敵意のある行為を行うが、良性に見えるような秘密のコードから構成される。通常は、疑いを避けるために贈り物や、時にはほんのわずかな価格で提供される。ユーザは、そのプログラ

ムの有用な機能のために、自分自身のシステムにそのプログラムを受け入れる。しかし、一旦ユーザのコンピュータ内に入ったら、そのプログラム中に隠されたコードが活性化し、悪意のある行為を行ったり、許可されていない人間がシステム資源へのアクセスを得ることができるように、システムセキュリティを破壊する方法を作り出す。トロイの馬攻撃は、秘密のコードによって行われる行為に依存して、受動的にもなるし能動的にもなる。例えば、秘密コードが単に情報を盗むだけなら、それは受動的である。しかし、ファイルを破壊したり汚染するような、より害悪のある行為を行う場合には、それは能動的である。

侵入者は、システムからのサービスを要求するクライアントに対して信頼できるサーバとして偽ることもできる。これは欺き (spoofing) として知られている。

11.2.2 能動的攻撃

能動的な侵入者は受動的なものよりも悪意があるものである。受動的な攻撃とは異なって、能動的な攻撃はシステムの正常な機能に干渉し、損害を残すことが多い。能動的な攻撃与える損害の中で最もよく見られるタイプは、ファイルの汚染、データの破壊、ハードウエア誤りの模倣、システム動作速度の低下、ごみデータによるメモリやディスクスペースの充満、システムのクラッシュ、偽造されたメッセージによる受信者の混乱、及びメッセージ配送の停止や遅延である。以下では、能動的な攻撃の通常の形態について述べる。なお、ウィルス、ウォーム及び論理爆弾に関する記述は、[Bowles and Pelaez 1992] に述べられている題材に基づいている。

ウィルス

コンピュータウィルスとは、合法的なプログラムに添付されたコードであり、そのコードが実行された時には、自分自身を複製して、他のプログラムに添付することによって感染するものである。この複製効果に加えて、ウィルスはシステムに対して、ファイルの汚染や消去などの損害を与えるのが普通である。したがって、その拡散していくという性質から、ウィルスはシステムに非常に大きな損害を与える。なお、ウィルスは能動的なタイプのトロイの馬攻撃である。

典型的なウィルスは以下のように動作する。侵入者は興味が湧き、有用な機能 (ゲームやユーティリティなど) を果たすプログラムを書き、それにウィルスを添付する。この添付は、そのプログラムが実行される時には、ウィルスコー

ドも一緒に実行されるような形でなされる。次に、侵入者はこの感染したプログラムを掲示板システムにアップロードしたり、そのシステムの他のユーザにメールしたり、無料又はフロッピーディスク代金だけの名ばかりの価格で提供したりする。もし誰かが感染したプログラムを使用すれば、そのウィルスコードが実行される。感染したプログラムのウィルスコードが実行された時には、それはハードディスク上の実行ファイルをランダムに選択し、それが既に感染しているかを確認する。大部分のウィルスはプログラムが感染していることを示すマーカーの役割を果たす文字列を含んでいる。もし選ばれたファイルが感染していれば、ウィルスは他の実行ファイルを選択する。感染していないプログラムが見つかれば、そのプログラムの最後に自分自身のコピーを添付し、そのプログラムの最初の命令をウィルスコードへのジャンプ命令で置き換えることによって、感染させる。ウィルスコードが実行を終了した時には、感染したプログラムが所期の機能を果たすように、その最初の命令であったものを実行し、次に 2 番目の命令にジャンプする。感染したプログラムが実行されるたびに、より多くのプログラムに感染しようとすることが原因で、ウィルスが蔓延することに注意する必要がある。また、オブジェクトファイルがそれ以上大きくなることを防ぐために、ウィルスは既に感染したプログラムには再度感染することがないことにも注意する必要がある。これによってウィルスは、ディスクスペース使用の顕著な増加なしに、多くのプログラムに感染することができる。

ウィルス感染からの回復は困難な仕事であり、攻撃されたコンピュータを部分的又は全面的に長時間シャットダウンする必要があることが多い。したがって、ウィルス問題を避けるために、必要な予防措置を取ることが望ましい。予防措置には、(a) 信頼できる店からのみソフトウェアを購入する、(b) 封を切られたソフトウェアや信用できない情報源から入手したソフトウェアの受け入れを拒否する、(c) 自分自身よりも安全基準が低いものからプログラムを借用しない、(d) パブリックドメインから無料で入手したソフトウェアや電子メールで送られてきたプログラムを掲示板にアップロードしない。

コンピュータシステムがウィルスに冒されたときには、治療しなければならない。最も簡単な治療方法は、シャットダウンし、主記憶とすべてのディスクをパージし、製造元のコピーを用いてゼロからファイルを再構築することである。殺菌ユーティリティも治療に使用可能である。このユーティリティは、まずよく知られたウィルスのマーカーと自身のマーカーを照合することによって、コンピュータがどのタイプのウィルスに感染したかを特定する。タイプが分かれば、そのウィルスが用いた感染方法の詳細な知識を駆使することによっ

て、感染した版からもとのプログラムを回復する。例えば、ホストプログラムの最初のジャンプ命令を書き換えるようなウィルスでは、回復はホストプログラムの開始点へジャンプするように戻すだけである。しかし、この殺菌ユーティリティは特定の知られたウィルス感染を治療できるだけであることに注意する必要がある。新しく出くわしたタイプのウィルスは治療できない。通常、優れた殺菌ユーティリティは数百タイプのウィルスを治療することができ、新しいウィルスの発見に対応して、頻繁に更新することによって、その能力を常に向上させることができる。

システムに長くウィルスが存在すれば、より多くの蔓延時間を持つことになり、回復がより困難になる。したがって、可能な限り早くウィルスを検出することが重要である。効果的なウィルス検出の方法は、スナップショットプログラムと確認ルーチンを用いることである。スナップショットプログラムは、初期導入時点で、すべての重要なシステム情報のログを採るために用いられる。確認ルーチンは、システムの現在の状態と元のスナップショットを比較するために、周期的に実行される。感染の兆候が検出されたら、コンピュータの感染領域を特定し、ユーザに通知する。

ウィルスに感染した分散システムを治療することは、さらに難しい問題である。感染をすべてのワークステーションから同時に除去しなければ、再感染が起こり得るからである。これは、ネットワークサーバ上の感染ファイルが、ネットワーク上のすべてのワークステーションを感染させ得るからである。

ウォーム

ウォームはコンピュータネットワーク内で、あるコンピュータから他のコンピュータへと広がっていくようなプログラムである。ウォームは、コンピュータネットワーク上で資源を共有する方法を利用して広がっていく。時には、ネットワークシステムに導入されている標準ソフトウェアの欠陥につけ込んで広がっていくこともある。ウォームプログラムは、あるネットワークノードに到着した後、破壊的な活動を行うかもしれない。直接的に破壊的でない場合でも、ネットワーク上のコンピュータの動作を本来の目的とは異なったものにしたり、資源を占有したり、ネットワークの通信リンクを飽和させたりすることによって、ネットワーク障害を引き起こす。ウォーム問題から回復するためには、システム全体 (ネットワーク上のすべてのコンピュータ) をシャットダウンしなければならない場合が多い。

ウォームプログラムがどのようにネットワークを伝搬していくかを示すため

に、コーネル大学大学院生 Robert Tappan Morris が 1988 年 11 月 2 日に起こした有名な Internet Worm 攻撃について述べる。この攻撃では、インターネット上の数千台もの UNIX マシンが感染させられた。このウォームプログラムは、2 つのタイプのコードを持つ。即ち、ブートストラップコードとウォームの本体を形成するコードである。ブートストラップコードは、攻撃対象のシステム上でコンパイルされ、実行される。実行された時、ブートストラップコードは新しいホストと、出身マシンとの間に通信リンクを確立し、ウォームの本体を新ホストにコピーし、それを実行する。一旦、新ホストに導入されれば、ウォームの最初の仕事は、その存在を隠すことである。そのために、自分自身のバイナリ版をアンリンクし、その親プロセスの強制終了、それ自身のファイルのメモリへの読み込みと暗号化を行う。次に、システムに入り込む際に生成されたファイルを削除する。このような隠蔽動作が完了した次の仕事は、そのホストのルーティングテーブルを覗いて、そのホストとその時点で接続されている他のホストの情報を収集することである。この情報を用いて、そのブートストラップコードをそれらのマシンにまき散らそうとする。このまき散らしは、以下の 3 つの方法を 1 つずつ試すことによって実行される。

1. 最初の方法は、UNIX の rsh コマンドを用いて、攻撃対象マシン上で遠隔シェルを立ち上げようとすることである。幾つかのマシンは他のマシンを信用しており、その結果、遠隔マシンの認証処理を行うことなく rsh を実行することから、この方法がうまく行く場合もある。もしうまく行けば、遠隔シェルはウォームプログラムを対象マシンにアップロードし、そこから新しいマシンへの拡散を継続する。

2. 最初の方法が失敗した場合には、2 番目の方法が試される。すべての BSD サイトでデーモンプロセスとして実行されている finger プログラムのバグを利用する方法である。インターネットワークの任意のユーザは、次のようにタイプすることによって、特定サイトの人物の一般情報を獲得することができる。

```
finger user_name@host_name
```

ここで一般情報とは、その人の本名、自宅住所、勤め先住所、電話番号などである。finger ユーティリティは 、入力データを読むのに、C ライブラリ関数 gets を使用する。gets の問題は、バッファオーバーフローを確認することなく、

入力ストリングをすべて読んでしまうことである。ウォームはこの欠陥につけ込んで、パラメータとして特別に構成された536バイトのストリングを与えて、fingerを呼び出す。オーバーフローによって、システムスタックの上書きが発生し、これによって、ウォームは自分に都合の良い命令 (/bin/shを実行する手続き) をスタックに置くことができる。要求を受け取った時点の手続きからfingerデーモンが戻った時には、mainに戻るのではなく、スタック上の536バイトのストリング中の手続きに戻り、その手続きを実行する。この方法がうまく行けば、ウォームは対象マシン上で走行するシェルを獲得したことになる。

3. 最初の2つの方法が失敗した場合には、UNIXの電子メールユーティリティであるsendmailのループホールを利用する第三の方法を試みる。sendmailプログラムはDEBUGオプションを持っており、それによってプログラムテスターは、メイラーのアドレス決定ルーチンを呼び出すことなく、メールがそのサイトに到着しているかどうかの確認を行うことができる。多くのベンダーやサイト管理者は、ローカルな状況に応じてメイラーを容易に再構成できるように、sendmailコードのデバッグオプションをそのまま残している。ウォームが行うことは、DEBUGオプションの付いたsendmailプログラムを実行し、ブートストラップコードを対象マシンにメールするためのコマンド列を実行させることである。

一旦新しいマシンに定住すると、ウォームはUNIXパスワードファイルへアクセスできることと、ユーザがパスワードをつける場合の共通の傾向を利用して、ユーザアカウントに侵入しようとする。破られたそれぞれのパスワードを用いることによって、ウォームはそのパスワードに対応するユーザになりすますことができ、そのユーザがアカウントを持つ任意の遠隔マシンにアクセスすることができる。

ウォームは、発見されるのを防ぐために、知的に振る舞うように設計されている。周期的に自分自身をforkし、親をkillする。その結果、ウォームのプロセスIDは常に変化する。これによって、1つのプロセスが大量のCPU時間を消費することを避けることができる。大幅なCPU時間消費によって、疑いを引き起こしたり、スケジューリング優先度の低下を招いたりする可能性がある。さらに、12時間毎に感染したマシンに保持されている自身の記録を消去し、既に感染したことがあるマシンを潜在的な攻撃対象のリストに戻す。ウォームが新しいマシンへのアクセス権を得た場合には常に、そのマシンが既にそのウォームのコピーを持っているかどうかを確認する。もし持っていれば、7回

の内 1 回を除いては、新しいコピーは処理を終了する。7 回に 1 回はコピーを無視するのは、恐らく、システム管理者が本物のウォームをだますために、偽のウォームを走らせているかもしれないようなマシンに対してさえも拡散していくことができるようにするためであろう。

ウィルスもウォームも共に自分自身を複製して拡散していくが、以下の点で異なっている。

1. ウィルスはプログラムの断片であるのに対して、ウォームはそれ自身で完全なプログラムである。

2. ウィルスはプログラム断片であるため、それ自身が独立して存在することはできない。ホストプログラムの中に存在し、そのホストプログラムが実行された時にだけ実行され、その存在はホストプログラムに依存している。一方、ウォームは独立に存在し、独立に実行可能なものである。

3. ウィルスは、あるプログラムから他のプログラムへ拡散していくが、ウォームは、ネットワーク内のあるコンピュータから他のコンピュータへ拡散していく。

論理爆弾

論理爆弾とは、何らかのトリガー条件がそれを爆発させるまでは、休止状態にあるようなプログラムである。爆発した時には、ホストコンピュータのデータを破壊し、そのシステムソフトウェアを駄目にしてしまう。トリガー条件は、特定ファイルへのアクセス、あるプログラムのある回数の実行、ある時間の経過、システム時計のある特定の日時への到達 (例えば、13 日の金曜日やエイプリルフール) などのイベントである。通常、トリガー条件はシステムに対して最大のダメージを与えることができるように選ばれる。論理爆弾はトロイの馬に組み込むこともできるし、ウィルスによって運ぶこともできる。

メッセージ通信に伴う能動的攻撃

分散システムでは、システム内のあるノードから他のノードへメッセージという形で情報を運ぶのに、通信チャネルが用いられる。通信チャネルは攻撃者にさらされており、攻撃者は通信チャネルを流れるメッセージを観察、修正、削除、挿入、遅延、回送、再送することによって、システムの安全性を損ねようとする。メッセージ通信に伴うよく知られた能動的攻撃には以下のタイプの

ものがある。

1. **完全性攻撃** (integrity attack):安全な通信のためには、すべてのメッセージは、送られたものと完全に同じものが受信されるか、矛盾があればそれが検出されるという完全性が要求される。しかし、侵入者が、通信チャネルを流れる間にメッセージの内容を変更し、受信者がその変更に気づかず、それを元のメッセージとして受け取ってしまうかもしれない。

2. **信憑性攻撃** (authenticity attack):侵入者が、自身のコンピュータを通信チャネルに違法に接続し、合法的なネットワークサイトになりすますかもしれない。この侵入者は、正しいアドレスを持った偽のメッセージを作成し、本物のメッセージとして配送されるように、それをシステムに挿入することができる。もし完全性攻撃が可能なら、メッセージが間違った宛先に配送されるように、メッセージのプロトコル制御情報 (アドレス) を変更することによって、信憑性攻撃も仕掛けることができる。

3. **拒絶攻撃** (denial attack):拒絶攻撃では、侵入者は 2 つのプロセス間の通信パスを完全にブロックし、これらが全く通信できないようにするか、又は 2 つのプロセス間で交換されるすべてのメッセージを観察し、選ばれたものだけを配送できないようにする。即ち、侵入者はメッセージ配送の完全なまたは部分的な拒絶を行う。

4. **遅延攻撃** (delay attack):メッセージには時間的な価値を持つものがある。したがって、拒絶攻撃の代わりに、2 つの通信プロセス間の連携において、メッセージの配送を単に遅らせるだけで、目的を達成できる場合もある。

5. **再送攻撃** (relay attack)。再送攻撃では、侵入者は古いメッセージを再送し、受信側ではそれを新しいメッセージとして受け取る。

暗号 (cryptography) は、機密データを理解できないように暗号化することであり、電話回線、マイクロ波通信、衛星通信などの不安全なチャネルを用いて送られる情報を保護する唯一の実用的な方法である。これは、暗号化されたメッセージは元のメッセージに関する情報を与えることがなく、その結果、機密性を保証するからである。また、暗号化されたメッセージは、不正を加えら

れた場合には、元のメッセージに正しく戻されることはなく、その結果、完全性が保証される。暗号は、通信し合う実体間の安全な身元確認にも用いることができ、信憑性が保証される。さらに暗号化は、プロトコルと一体となって、メッセージの拒絶、遅延、再送を防ぐために用いることもできる。例えば、古いメッセージの再送には、ノンス又はタイムスタンプを用いて、対処することができる。**ノンス** (nonce) は、常に最新であることが保証された情報である。即ち、それ以前に用いられたり現れたりしないものである。したがって、最近に送られたノンスに依存した機能を含むような応答は、時間的に正しいものと考えられる。その応答は、そのノンスが送られた後にのみ生成することができたはずだからである。ノンスとして用いるには、完全な乱数が相応しい。要約すれば、メッセージ通信に伴う攻撃を防ぐ唯一の方法は暗号技術を適用することである。

11.2.3　監禁問題

クライアントサーバモデルでは、1 つのサーバプログラムが複数のクライアントプログラムに共有される場合がある。プログラムが共有されるような状況では、データオブジェクトだけが共有される場合に比べて、セキュリティの問題は極めて難しくなる。1 つの理由は、既に見てきたトロイの馬である。トロイの馬は、共有プログラムが機密情報を資格を持たない対象者に漏らすことができるような 1 つの方法である。共有プログラムが秘密情報を許可されていない対象者に漏らすことができる方法は他にも幾つかある。

秘密情報を保持することがなく、その結果、漏らすこともできないプログラムは、**無記憶** (memoryless) 又は**監禁状態** (confiend) と言われ、漏洩を防止することは**監禁問題** (confinement problem)[Lampson 1973] と呼ばれる。即ち、監禁問題とは、許可された対象者がアクセス権を持つオブジェクトに含まれる情報を、その情報へのアクセスが許可されていない対象者へリリースするすべての方法を除去するという問題を扱うものである。Lampson によれば、プログラムが情報を保持せず、出力もしない限り、監禁はそのプログラムへのアクセス権を制限するだけで実現することができる。しかし、プログラムが情報を保持したり、出力しなければならない場合には、安全性を保証するにはアクセス制御だけでは不十分である。

Lampson はプログラムが情報を漏洩するのに用いることができるチャネルを以下のように特定している。

1. **合法的チャネル** (legitimate channel):合法的チャネルとは、メッセージやプリントアウトのように、プログラムがその処理結果を運搬するために用いるチャネルのことである。プログラムは、実際の処理結果と並んで、付加的な情報をその中に隠蔽することができる。その付加的な情報を運ぶためには、結果を受け取った人ないしはプロセスにとって意味があるような何らかの符号化手段が用いられる。例えばプリントアウトにおいて、普通の人には識別できないが、詳細に観察すれば分かるような、単語間の長さの異なる2種類の空白を、0又は1を意味するように用いることができる。このようなタイプのプリントアウトは、2つの単語間の隠されたビットについて知っている人に対して、付加的な情報を送るために用いることができる。

2. **記憶チャネル** (storage channel):記憶チャネルとは、他のプロセスに情報を漏洩するのに、共有変数や共有ファイルなどのシステム記憶を用いるものである。記憶チャネルを用いて、プロセスAがプロセスBに情報を漏らしたい時に、必ずしも、双方のプロセスがその共有オブジェクトへのアクセス権を持つ必要はないことに注意が必要である。例えば、システムがファイルをロックする機構を備えている場合には、プロセスAは1を表すためにあるファイルをロックし、0を表すためにそのファイルをアンロックすることができる。プロセスBが、アクセス権を持たないようなファイルのロック状態を知ることが可能な場合もある。同様に、UNIXでは、プロセスAは1を表すためにファイルを生成し、0を表すためにそのファイルを削除することもできる。プロセスBがプロセスAによって生成されたファイルへのアクセス権を持たなくとも、accessシステム呼び出しを用いて、そのファイルが存在するかどうかの確認を行うことはできる。

3. **秘密チャネル** (covert channel):秘密チャネルとは、通常は情報転送を目的としないが、ある種の情報を送るためには使用できるようなパスのことである。例えば、以下のような方法によって、あるプロセスが、ある種の活動を注意深く観察する他のプロセスに対して、情報を漏らすことができる [Tanenbaum 1992]。

■ページング頻度の変更。例えば、ある一定時間内に、そのプロセスが多くのページ例外を起こす状態を1、全くページ例外が起こらない状態を0として

用いることができる。

■ CPU 使用率の変更。例えば、ある一定時間内の CPU を使用している状態を 1、同じ期間内にプロセスが休眠している状態を 0 として用いることができる。

■特定の資源の獲得と解放。例えば、プロセスがその資源を獲得している状態を 1、解放している状態を 0 としても用いることができる。

監禁問題を解決するためには、あるプログラムが他のプログラムと通信するすべてのチャネルをブロックすることが重要である。しかし、このようなチャネルをすべて見つけ、ブロックしようとすることは極めて困難である。現実的には、できることはほとんどない。したがって、監禁問題は、一般的には解決不可能である。

11.3　暗号

暗号とは、物理的なセキュリティを提供することが困難な状況で、許可されていないアクセスから個人情報を保護するための手段である。このセキュリティ技術の背景にある基本的な考え方は、情報のコピーを防ぐことができないなら、その理解を妨げた方が良いということである。

11.3.1　基本コンセプトと用語

暗号技術において用いられる 2 つの基本操作は**暗号化** (encryption) と**復号化** (decryption) である。暗号化 (英語としては enciphering という用語も用いられる) とは、判読可能な情報 (**平文** (plaintext 又は cleartext) と呼ばれる) を判読不可能な形態 (**暗号文** (ciphertext) と呼ばれる) に変換する処理のことである。復号化 (英語としては deciphering という用語も用いられる) とは、情報を暗号文から平文に戻す処理のことである。通信チャネルを通して転送される情報に暗号技術が使用される場合には、平文のことを**メッセージ** (message) と呼ぶ。

基本的に暗号化とは、以下のような形式を持つ数学的な関数 (暗号化アルゴリズム) である。

$$C = E(P, K_e)$$

ここで、P は暗号化されるべき平文、K_e は暗号化鍵、C は結果的に得られる暗号文である。C の復号化は、以下の形式を持つ関数 (復号アルゴリズム) によって行われる。

$$P = D(C, K_d)$$

ここで、K_d は復号鍵である。復号化関数 D は暗号化関数 E の逆関数である。即ち、以下の関係が成立する。

$$D(E(P, K_e), K_e) = P$$

平文が簡単に知られてしまうことを防ぐために、ある平文を、特定のパラメータで選ばれ得る暗号文の大きな集合に変換することが可能でなければならない。鍵 K_e と K_d がこのパラメータの役割を果たす。即ち、関数部分が同じであっても、鍵は必要に応じて何度も変更される。

上述のような、暗号システムに関する一般的な構造の記述は、図 11.1 のように図示される。図において、メッセージは不安全なチャネルを通して送信ノードから受信ノードへの安全な伝送を行うために暗号化される。

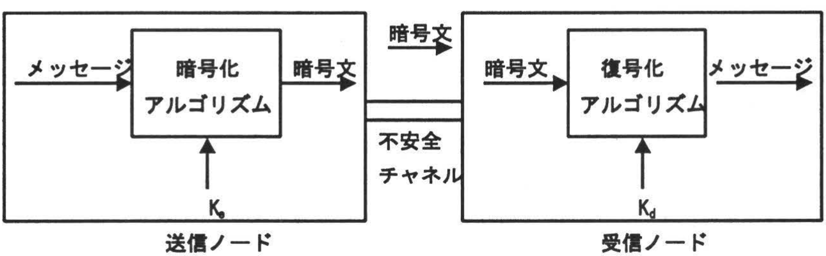


図 **11.1** 暗号システムの一般的構造

11.3.2　基本的な要求

実用的なものとなるためには、暗号システムは以下の基本要求を満たさなければならない。

1. 使い易くなければならない。暗号化/復号化アルゴリズムはコンピュータが効率よく実行できるものでなければならない。

2. セキュリティを達成するには 2 つの方法がある。まず、暗号化アルゴリズムを秘密にし、それを推測することさえ困難な状況にしておく方法がある。次に、暗号化アルゴリズムは公開するが、鍵は秘密で、現実的には推測不可能なくらいに長くしておく方法である。実用システムでは二番目の方法が好まれている。即ち、システムのセキュリティは鍵の秘密性のみに依存すべきであり、アルゴリズムの秘密性に依存すべきではない。

3. システムは計算論的な意味で (実用的な意味で) 安全でなければならない。即ち、攻撃者 (**暗号解読者** (cryptanalyst) とも呼ばれる) にとって、K_d の決定が計算論的な意味で実行不可能でなければならない。暗号システムの強度は K_d を決定する難しさ (通常は時間や基本操作の回数で測られる) の程度を尺度とする。暗号システムでは、侵入者が利用可能な情報の量に応じて、3 つのタイプの攻撃が考えられる。即ち、暗号文攻撃、既知平文攻撃及び選択平文攻撃である [Bright 1977]。

暗号文攻撃 (ciphertext-only attack) では、侵入者は暗号文のみを傍受することができ、その暗号文から K_d を求めようとする。暗号文攻撃に耐えられないようなシステムは、完全に不安全と考えられ、無用のものである。

既知平文攻撃 (known-plaintext attack) では、侵入者はかなりの量の暗号文とそれに対応する平文を入手し、それらから K_d を求めようとする。既知平文攻撃に耐えられるシステムは安全と考えられる。

選択平文攻撃 (chosen-plaintext attack) では、侵入者は選択した任意の平文に対応する暗号文を得ることができる。注意深く選んだ平文に対する暗号文を調べることによって、K_d を求めようとする。今日では、選択平文攻撃に耐えられる能力によって暗号システムを評価することが最も適当である。

暗号システムが安全であることを示すもう 1 つの重要な方法は、時間によるテストである。システムが公開されれから、攻撃の成功が報告されておらず、十

分な期間 (年単位で測られる) 使用されてきたなら、その暗号システムはかなり優れたセキュリティを提供すると考えられる。

11.3.3 対称暗号システムと非対称暗号システム

暗号システムには、対称暗号システムと非対称暗号システムという2つの大きなクラスがある。**対称暗号システム** (symmetric cryptosystem) では、暗号化鍵 (K_e) と復号化鍵 (K_d) が同じであるか、又は一方から他方が容易に求められる。通常、共通の鍵 (K) が暗号化と復号化の両方に用いられる。セキュリティの観点からは、対称暗号システムの鍵は容易に変更でき、常に秘密に保たれていることが重要である。このことから、鍵は許可されたユーザだけに知らされる。対称暗号システムは、**共有鍵暗号システム** (shared-key cryptosystem) 又は**私有鍵暗号システム** (private-key cryptosystem) としても知られている。

対称暗号システムは、暗号化と復号化が信頼されたサブシステムによって行われるような状況で有用である。例えば、パスワードに基づくユーザ認証システムでは、この方式を使用することによって、暗号化形式のパスワードを保存しておくことができる。ユーザがパスワードを宣言した時には、オペレーティングシステムは暗号化鍵を用いてそのパスワードを暗号化し、その結果を内部的に格納する。認証段階では、オペレーティングシステムは同じ鍵を用いて、格納されているパスワードを復号し、ユーザによって与えられたパスワードと比較する。

一方、**非対称暗号システム** (asymmetric cryptosystem) では、復号化鍵 (K_d) と暗号化鍵 (K_e) は同じものではない。さらに、K_e から K_d を求めることは、計算論的な意味で現実的ではない。この特性によって、K_d だけを秘密にしておく必要があり、K_e は公開される。非対称暗号システムは**公開鍵暗号システム** (public-key cryptosystem) としても知られている。

公開鍵暗号システムは、計算論的に高価であり、大量データの暗号化には適していない。分散システムにおける公開鍵暗号システムの典型的な用途は、2つの通信実体 (A と B) の間で対称暗号システムを用いたメッセージ交換を行うために、これらの実体間の接続を確立する部分への適用である。A と B が接続を確立し、鍵 K の対称暗号システムによるメッセージ交換を開始したいと仮定する。A と B の間で鍵 K を共有するために、それを普通の通信チャネルを通して送ることは不安全であることに注意する必要がある。したがって、まず公開鍵暗号システムを用いて、A と B の間の接続を以下のように確立する。

■実体 A は公開鍵暗号システムの暗号化鍵 K_e を例えば電子掲示板などに掲載し、B がそれを獲得できるようにする。

■実体 B は A の公開鍵を用いて、鍵 K を暗号化し A に送信する。

■実体 A は、公開鍵暗号システムの復号化鍵 (K_d) を用いて、B のメッセージを復号する。K_d は A にのみ使用可能なので、このメッセージは A のみが復号できる。

■ここで A も鍵 K を持ったため、A と B は対称暗号システム方式を用いて、お互いに安全に通信することができる。盗聴者は公開鍵 (K_e)、暗号化された鍵 K および A と B の間で通信される暗号化されたメッセージしか知ることができない。

1 つの落とし穴は、誰かが A 又は B になりすますことである。この問題は、デジタル署名 (本章の後半で説明する) を用いることによって克服できる。

対称暗号システムと非対称暗号システムの相対的な利点／欠点は以下の通りである。

1. 対称暗号システムは暗号化と復号化が信頼できるサブシステムで実行されることを必要とするが、非対称暗号システムではその必要はない。したがって、一般的なセキュリティ方針では、非対称暗号システムが必要となる。

2. 通信システムにおけるメッセージのセキュリティに用いられた場合、対称暗号システムでは、送信者が暗号化に用いられる鍵を受信者に教えるための安全なチャネルを必要とする。しかし、暗号化されたメッセージは不安全なチャネルを通して送られるかもしれない。一方、非対称暗号システムでは、公開鍵とメッセージの両方を不安全なチャネルを通して送ることができる。したがって、鍵を送るための特別な安全チャネルを用意する必要がない。この理由から、非対称暗号システムは、対称暗号システムよりも安全であると考えられている。

3. 一般的に、非対称暗号システムの計算負荷は対称暗号システムよりも大きく、大量データの暗号化においては効率的ではない。その結果、実際のデータ通信では、非対称暗号システムが使われるのは初期化や制御機能においてのみである。一方、対称暗号システムは実際のデータ転送に用いられる。

データ暗号化標準 (Data Encryption Standard)(DES)[NBS 1977, Seberry and Pieprzyk 1989] は、今日最もよく知られ、広く用いられている対称暗号システムである。一方、Rivest - Shamir - Adleman(RSA) 暗号システム [Rivest et al. 1978, Seberry and Pieprzyk 1989] は、最初に公開され、最も満足できる非対称暗号システムである。

11.3.4 鍵配布問題

分散システムにおける安全な通信のために暗号技術を用いる場合、鍵配布の必要が生じる。これは、2 つの通信実体が安全に通信できるのは、送信メッセージの暗号化や復号化に用いられる鍵を獲得した場合に限られるからである。2 つの通信実体によって保持される、互いに対応した鍵の組によって、これらの実体間に独立した、私的な論理チャネルが構成される。鍵配布問題は、このような論理チャネルを生成するために必要な鍵を安全に提供する方法を扱うものである。対称暗号システムと非対称暗号システムにおける鍵配布問題は、それぞれ以下のように記述される。

対称暗号システムにおける鍵配布

2 つの異なるノード上のユーザ (人又はプログラム) が対称暗号システムを用いて安全な通信を行いたい場合、まず暗号化鍵と復号化鍵を共有しなければならない。そのためには、一方のユーザから他方に鍵を転送しなければならない。しかし、鍵を転送するための特別な伝送媒体はなく、メッセージ交換が行われるのと同じ不安全な物理媒体を用いて鍵を転送しなければならない。そのため、転送前に鍵自身を暗号化する必要がある。これは、侵入者が不安全な媒体中を転送中の鍵を獲得した場合、2 つにユーザ間で交換されるすべての暗号化メッセージを復号できるからである。したがって、対称暗号システムには循環性が存在する。この循環性は何らかの安全な手段で、少数の鍵を事前配布することによって解決される。よく用いられる方法は、**鍵配布センター** (key distribution center)(KDC) の役割を果たすサーバプロセスを用いることである。システムの各ユーザは、あらかじめ用意されたユニークな鍵の組を KDC と共有する。

一般に、KDC は信頼できる実体であり、システムのすべての通信ユーザによって共有される。ユーザの要求に応じて、新しい秘密鍵を生成し、それを用いてそのユーザは他のユーザと通信を行う。実際の実装では、システム内に複

数のKDCがあってもよい。以下に示す3つの実装方式がよく用いられる。

■集中型方式
■完全分散型方式
■部分的分散型方式

以下では、各方式において、安全な通信を行いたい2つのユーザ間でどのように鍵配布が行われるかを述べる。

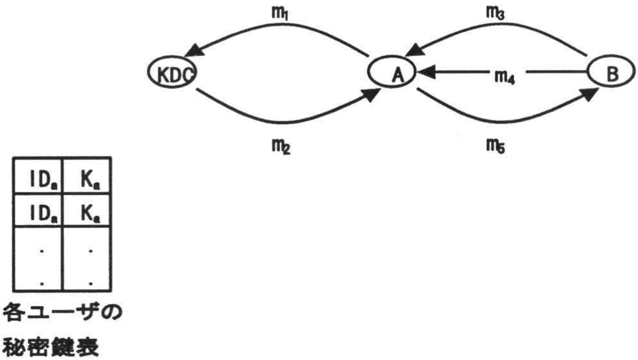
集中型方式 (centralized approach)

この方式では、単一の集中型KDCが用いられ、それが各ユーザの秘密鍵の表を管理する (図11.2を参照)。あるユーザの秘密鍵はそのユーザとKDCだけが知っている。ユーザAとBの秘密鍵をそれぞれ K_a と K_b とし、それを用いてAとBの間に暗号メッセージ交換のために安全な論理チャネルが確立されると仮定する。これを実現するために、以下のプロトコルが [Needham and Schroeder 1978] によって提案されている (図11.2を参照)。

1. ユーザAは要求メッセージ (m_1) をKDCに送り、ユーザBとの間で安全な論理チャネルを確立したいことを表明する。そのメッセージは要求コード (R_a)、Aのユーザ識別子 (ID_a) 及びBのユーザ識別子 (ID_b) を含む。このメッセージはユーザAからKDCに対して、平文の形で転送される。

2. メッセージ m_1 を受信した時、KDCは表検索により K_a と K_b を求める。K_a と K_b はそれぞれメッセージ中のユーザ識別子 ID_a と ID_b に対応している。次にKDCはAとBの間の安全な通信のための秘密鍵 K_{ab} を生成する。KDCは、鍵 K_b を用いて、組 (K_{ab}, ID_a) を暗号化し、暗号文 $C_1 = \mathrm{E}((K_{ab}, ID_a), K_b)$ を生成する。最後にKDCは R_a、ID_a、K_{ab} および C_1 を含むメッセージ (m_2) をユーザAに送信する。メッセージ m_2 は、ユーザAだけが復号できるように、鍵 K_a で暗号化される。

3. メッセージ m_2 を受信した時、ユーザAは鍵 K_a でそれを復号し、メッセージ中の R_a と ID_a が元のそれと同じかどうかを調べて、m_2 が m_1 への応答であることを確認する。もし正しければ、ユーザAは鍵 K_{ab} を将来の使用に備えて保持し、ユーザBにメッセージ m_3 を送る。このメッセージは暗号文



m_1 = (Ra, IDa, IDb)
　ここで、Ra = ユーザAによってなされた要求のコード
　　　　　IDa = ユーザAの識別子
　　　　　IDb = ユーザBの識別子

m_2 = E((Ra, IDa, Kab, C1), Ka)
　ここで、Kab = ユーザAとBの間の安全な通信のためにKDCが生成した秘密鍵
　　　　　C1 = E((Kab, IDa), Kb)
　　　　　　ここで、Kb = ユーザBの固有鍵
　　　　　Ka = ユーザAの固有鍵

m_3 = C1

m_4 = C2 = E(Nr, Kab)
　ここで、Nr = ユーザBによって生成された乱数

m_5 = C3 = E(Nt, Kab)
　ここで、Nt = f(Nr)であり、fはあらかじめ定められた関数

図 **11.2**　集中型方式における鍵配布方法

C_1 を含む。C_1 は鍵 K_b を用いて生成されているので、ユーザ B だけが復号できることに注意する必要がある。

4. メッセージ m_3 を受信した時、ユーザ B は鍵 K_b で C_1 を復号し、K_{ab} と ID_a の両方を獲得する。この段階で、ユーザ A と B は共に同じ鍵 K_{ab} を持っている。他のユーザがこの鍵を持つことはないから、K_{ab} は A と B の間の安全な通信に使用することができる。この時点で、ユーザ B は、ユーザがの鍵 K_{ab} を持っているかどうかを確認する必要がある。したがって、ユーザ B は認証処理を開始する。この認証処理は、ユーザ A にノンスを送り、最近送られたノンスの何らかの関数を含んだ応答を受信することから構成される。このために、ユーザ B は乱数 N_r を生成し、それを鍵 K_{ab} で暗号化して、暗号文 $C_2 = \mathrm{E}(N_r, K_{ab})$ を生成し、これをメッセージ m_4 に入れてユーザ A に送信する。乱数 N_r はノンスとして使用される。

5. メッセージ m_4 を受信すると、ユーザ A は鍵 K_{ab} で C_2 を復号し、N_r を取り出す。次に、N_r をあらかじめ定められた関数 (f) で変換して、新しい値 N_t を求める。ユーザ A は鍵 K_{ab} で N_t を暗号化して、暗号文 $C_3 = \mathrm{E}(N_t, K_{ab})$ を生成し、これをメッセージ m_5 に入れてユーザ B に送信する。

6. メッセージ m_5 を受信すると、ユーザ B は C_3 を復号して、N_t を取り出し、それに f の逆関数を適用して選られる値が N_r に等しいかどうかを確認する。もし等しければ、ユーザ B は A と B の間に鍵 K_{ab} を用いた安全なチャネルが生成されたことを確認したことになる。これは、相互信頼を実現するのに十分であり、これ以降ユーザ A と B の間では、鍵 K_{ab} によって暗号化されたメッセージ交換を行うことができる。

このプロトコルには問題があることが Denning と Sacco[1981] によって指摘された。メッセージ m_3 の転送途上で、侵入者が C_1 をコピーし、不特定の方法で K_{ab} を知った場合、その侵入者は、それ以降、B に対して A になりすますことができる。ここでの基本的な問題は、KDC に対して B がノンスを提供する機会がなく、その結果、KDC から得られる K_{ab} の新鮮さを推定する手段を持たないことにある。KDC とユーザ A のみが K_{ab} が最新であることを知っている。この問題点を修正する方法の 1 つは、暗号文 C_1 にタイムスタンプ (T) を付加し、$E((K_{ab}, ID_a, T), K_b)$ という形にすることである。ユーザ B はこ

のメッセージを復号し、T が最新であるかどうかを確認する。この解決策はケルベロスシステム (本章の後半で説明する) で採用されている。

集中型 KDC 方式では、システムのユーザ数が n であれば、安全な通信を提供するためには、あらかじめ用意された n 個の鍵の組が必要である。この方式は実装するのは容易であるが、単一の KDC では、低い信頼性と性能ボトルネックの問題を持つ。即ち、KDC が存在するノードがクラッシュした場合には、新鮮な鍵の配布が行えなくなり、大規模システムではユーザの数が多くなるにつれて、KDC が過負荷となる。以下で説明する 2 つの方式は、集中型方式におけるこのような欠点を克服するために用いることができる。

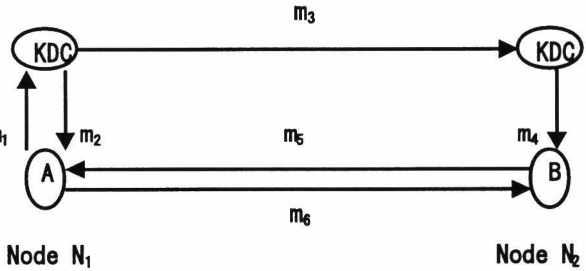
完全分散型方式 (fully distributed approach)

この方式では分散システムの各ノードに 1 つの KDC が配置される。事前に秘密鍵を配布することによって、各 KDC は他のすべての KDC と安全に通信することができる。即ち、各 KDC は他のすべての KDC の固有鍵から構成される秘密鍵表を持っている。したがって、n ノードから構成されるシステムでは、各 KDC は $(n-1)$ 個の鍵を保持し、システム全体では $n(n-1)/2$ 組の鍵が保持されることになる。

ここで、ノード N_1 のユーザ A とノード N_2 のユーザ B の間で安全な論理通信チャネルを確立すると仮定する。また、K_1 と K_2 をそれぞれノード N_1 とノード N_2 の KDC の固有鍵と仮定する。所望の接続は以下のような手順で確立することができる (図 11.3 を参照)。

1. ユーザ A は要求メッセージ (m_1) をローカルな KDC に送る。このメッセージは、要求コード (R_a)、A のユーザ識別子 (ID_a) 及び B のユーザ識別子 (ID_b) を含む。すべてのローカル通信は安全であり、そのためローカルなメッセージは平文で安全に転送されると仮定している。

2. メッセージ m_1 を受信すると、ノード N_1 の KDC は名前サーバに問い合わせて、識別子 ID_b を持つユーザのノード N_2 の位置を得る。そして、自身が管理する表からノード N_2 の KDC の固有鍵 (K_2) を引き出す。次に、ユーザ A と B の間の安全な通信のための秘密鍵 K_{ab} を生成する。鍵 K_2 を用いて、(K_{ab}, ID_a, ID_b) を暗号化し、暗号文 $C_1 = \mathrm{E}((K_{ab}, ID_a, ID_b), K_2)$ を生成する。最後に、R_a、ID_a、K_{ab} を含むメッセージ (m_2) をユーザ A に送り、C_1 を含むメッセージ (m_3) をノード N_2 の KDC に送る。



m_1 = (Ra, IDa, IDb)

ここで、Ra = ユーザ A によってなされた要求のコード

IDa = ユーザ A の識別子

IDb = ユーザ B の識別子

m_2 = (Ra, IDa, Kab)

ここで、Kab = ユーザ A と B の間の安全な通信のためにノード N1 の KDC が生成した秘密鍵

m_3 = C_1 = E((Kab, IDa, IDb), K2)

ここで、K2 = ノード N2 の KDC の固有鍵

m_4 = (Kab, IDa)

m_5 = C_2 = E(Nr, Kab)

ここで、Nr = ユーザ B によって生成された乱数

m_6 = C_3 = E(Nt, Kab)

ここで、Nt = f(Nr) であり、f はあらかじめ定められた関数

図 11.3　完全分散方式における鍵配布方法

3. メッセージ m_2 を受信すると、ユーザ A はメッセージ内の R_a と ID_a が元のものと同じかどうかを調べ、m_2 が m_1 に対する応答であることを確認する。もしそうであれば、ユーザ A は将来の使用に備えて、鍵 K_{ab} を保存しておく。

4. 一方、メッセージ m_3 を受信すると、ノード N_2 の KDC は鍵 K_2 でそれを復号し、識別子 ID_b を持つユーザに対して (K_{ab}, ID_a) をメッセージ m_4 の形で転送する。

5. メッセージ m_4 を受信すると、ユーザ B は認証手続きを開始し、集中型方式を全く同じ方法で、ユーザ A の認証を行う。

この方式では、安全な通信を提供するために必要な、あらかじめ用意される鍵組の数は、ユーザ数とは独立であり、システム内のノード数のみに依存する。この方式のもう 1 つの利点は、2 ユーザ間の安全な通信のための鍵をうまく配布するために、この 2 ユーザのノードのみがうまく機能すればよいことである。したがって、この方式は極めて信頼性が高い。

部分的分散型方式 (partially distributed approach)

この方式では、システム内のノードが領域に分割され、各ノードに KDC を持つ代わりに、各領域に 1 つの KDC を配置する。この KDC はその領域内の 1 つのノードに配置される。秘密鍵を事前に配布しておくことによって、各 KDC は自領域内の各ユーザと安全な通信を行うことができるとともに、他のすべての領域の KDC とも安全な通信を行うことができる。即ち、各 KDC は、自領域内のすべてのユーザの固有鍵と他のすべての KDC の私有鍵からなる秘密鍵表を持つ。

この方式では、2 つのユーザ A、B の間の安全な論理通信チャネルの確立に関する鍵配布は、これら 2 ユーザの位置に依存する。もしユーザ A と B が共に同じ領域に属するノードにあるなら、鍵は集中型方式の場合と全く同じ手順で配布される。この場合、その領域の KDC が集中型 KDC の役割を果たす。一方、ユーザ A と B が異なる領域に属するノードにある場合には、鍵配布は完全分散型方式の場合と同様の手順で実行される。唯一の相違は、この場合には、メッセージ m_2 と m_4 も暗号化されることであり、これはそれらがあるノードから他のノードへ転送されるからである。即ち、m_2 はユーザ A の固有

鍵で暗号化され、m_4 はユーザ B の私有鍵で暗号化される。この方式における完全な鍵配布手順を図 11.4 に示す。この方式では、特定の領域の KDC が故障することは、その領域内のユーザを含む鍵配布活動を途絶させるだけである。他の領域のユーザに関する鍵配布はうまく実行可能である。したがって、この方式の信頼性は、集中型と完全分散型の間に位置する。

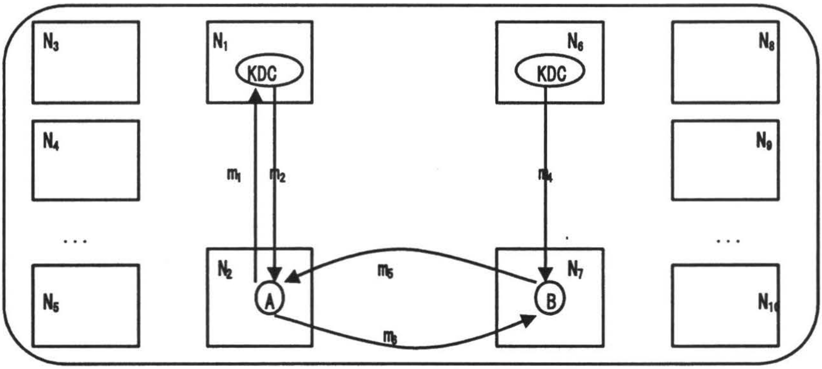
非対称暗号システムにおける鍵配布

非対称暗号システムでは、公開鍵のみが配布される。公開鍵は秘密にする必要がないので、不安全な物理媒体を通して伝送しても問題はない。したがって、非対称暗号システムでは鍵配布問題が存在しないように見える。しかし、非対称暗号システムの安全性はユーザによって正しい公開鍵が選ばれることに大きく依存しているので、この見方は正しくない。公開鍵を受け取ったユーザはそれが本物であることを確実にしたい。したがって、非対称暗号システムの鍵配布には認証手続きが含まれている。その認証によって、侵入者が鍵の組を生成し、あるユーザとの論理通信チャネルを確立するために、公開鍵をそのユーザに送ることを防ぐ。認証手続きによって、論理通信チャネルの 2 つのユーザが、実際のデータ交換を始める前に、身元を保証することができる。

非対称暗号システムでよく用いられる鍵配布方式は、**公開鍵管理者** (public-key manager)(PKM) プロセスを用いるものである。PKM はシステムに参加する全ユーザの公開鍵のディレクトリを維持管理する。PKM にも鍵の組 (P_k, S_k) が存在する。公開鍵 P_k はすべてのユーザに知られており、秘密鍵 S_k は PKM のみが知っている。以下では、この方式に基づいて 2 ユーザ間で論理通信チャネルを確立するためのプロトコルを説明する。

ユーザ A と B の間に論理通信チャネルを確立するものとする。また、(P_a, S_a) をユーザ A の鍵の組、(P_b, S_b) をユーザ B のそれとする。公開鍵 P_a と P_b は PKM に格納され、秘密鍵 Sa と S_b はそれぞれユーザ A と B のみが知っている。所望の接続は以下の手順で確立することができる (図 11.5 を参照)。

1. ユーザ A は PKM に要求メッセージ (m_1) を送り、ユーザ B との安全な論理通信チャネルを確立するように要請する。そのメッセージは、要求コード (R_a)、タイプスタンプ (T_1)、A のユーザ識別子 (ID_a) 及び B のユーザ識別子 (ID_b) を含む。このメッセージは P_k を用いて暗号化される。即ち、m_1 は暗号文 $C_1 = E((R_a, T_1, ID_a, ID_b), P_k)$ を含む。



領域 R1　　　　領域 R2

上に示すように、システムに 10 個のノードがあり、それらが 2 つの領域 R1 と R2 に分割されている。
領域 R1 と R2 の KDC はそれぞれノード N1 と N6 に配置されている。

m_1 = (Ra, IDa, IDb)

ここで、Ra = ユーザ A によってなされた要求のコード

IDa = ユーザ A の識別子

IDb = ユーザ B の識別子

m_2 = C_1 = E((Ra, IDa, Kab), Ka)

ここで、Kab = ユーザ A と B の間の安全な通信のために領域 R1 の KDC が生成した秘密鍵

Ka = ユーザ A の固有鍵

m_3 = C_2 = E((Kab, IDa, IDb), K2)

ここで、K2 = 領域 R2 の KDC の固有鍵

m_4 = C_3 = E((Kab, IDa), Kb)

ここで、Kb = ユーザ B の固有鍵

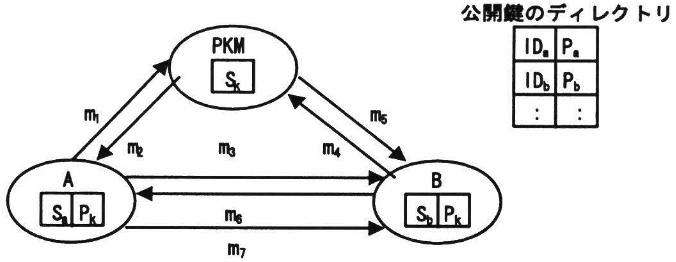
m_5 = C_4 = E(Nr, Kab)

ここで、Nr = ユーザ B によって生成された乱数

m_6 = C_5 = E(Nt, Kab)

ここで、Nt = f(Nr) であり、f はあらかじめ定められた関数

図 11.4　部分的分散方式における鍵配布方法



Pk = PKM の公開鍵　　Pb = ユーザ B の公開鍵
Sk = PKM の秘密鍵　　Sb = ユーザ B の秘密鍵
Pa = ユーザ A の公開鍵　　IDa = A のユーザ識別子
Sa = ユーザ A の秘密鍵　　IDb = B のユーザ識別子

$m_1 = C_1 = E((Ra,\ T1,\ IDa,\ IDb),\ Pk)$

　ここで、Ra = ユーザ A によってなされた要求のコード

　　T1 = タイムスタンプ

$m_2 = C_2 = E((Ra,\ T1,\ Pb),\ Pa)$

$m_3 = C_3 = E((IDa,\ Na),\ Pb)$

　ここで、Na = ユーザ A によって生成された乱数

$m_4 = C_4 = E((Pb,\ T2,\ IDa,\ IDb),\ Pk)$

　ここで、Rb = ユーザ B によってなされた要求のコード

　　T2 = タイムスタンプ

$m_5 = C_5 = E((Rb,\ T2,\ Pa),\ Pb)$

$m_6 = C_6 = E((Na,\ Nb),\ Pa)$

　ここで、Nb = ユーザ B によって生成された乱数

$m_7 = C_7 = E(Nb,\ Pb)$

図 11.5　非対称暗号システムにおける鍵配布方法

2. メッセージ m_1 を受信すると、PKM は P_k を用いて C_1 を復号する。次に、メッセージ内のユーザ識別子 ID_a と ID_b にそれぞれ対応する公開鍵 P_a と Pb を表から検索する。PKM は P_a を用いて、(R_a, T_1, P_b) を暗号化し、暗号文 $C_2 = E((R_a, T_1, \mathrm{Pb}), P_a)$ を生成する。最後に、メッセージ m_2 として、C_2 をユーザ A に送信する。

3. メッセージ m_2 を受信すると、ユーザ A は S_a を用いて C_2 を復号し、その内容を取り出す。取り出された R_a と T_1 は元の値と比較され、m_2 が m_1 への応答であり、古いメッセージの再送でないことを確認する。次に、ユーザ A は乱数 N_a を生成し、ユーザ B の公開鍵 P_b を用いて、(ID_a, N_a) を暗号化して、暗号文 $C_3 = E((ID_a, N_a),P_b)$ を生成する。この C_3 を、メッセージ m_3 としてユーザ B に送る。

4. メッセージ m_3 を受信すると、ユーザ B は S_b を用いて C_3 を復号し、その内容を取り出す。メッセージ中の ID_a を知ることによって、ユーザ B はユーザ A が自分との論理通信チャネルの確立を望んでいることを知る。ユーザ A の認証された公開鍵を獲得するために、ユーザ B は PKM に連絡する。そのためには、ユーザ B はメッセージ m_4 を PKM に送り、ユーザ A の公開鍵を要求する。このメッセージは要求コード (R_b)、タイムスタンプ (T_2)、A のユーザ識別子 (ID_a) 及び B のユーザ識別子 (ID_b) を含む。このメッセージは P_k を用いて暗号化される。即ち、m_4 は暗号文 $C_4 = E((R_b, T_2, ID_a, ID_b), P_k)$ を含む。

5. メッセージ m_4 を受信すると、PKM は P_k を用いて C_4 を復号する。次に、メッセージ中のユーザ識別子 ID_a と ID_b にそれぞれ対応する公開鍵 P_a と P_b を検索する。PKM は、鍵 P_b を用いて (R_b, T_2, P_a) を暗号化し、暗号文 $C_5 = E((R_b, T_2, P_a), P_b)$ を生成する。最後に、C_5 をメッセージ m_5 としてユーザ B に送る。

6. メッセージ m_5 を受信すると、ユーザ B は S_b を用いて C_5 を復号して、その内容を取り出し、R_b と T_2 を元のものと比較して、m_5 が間違いなく m_4 に対する応答であることを確認する。次に、ユーザ B は乱数 N_b を生成し、鍵 P_a を用いて (N_a, N_b) を暗号化して、暗号文 $C_6 = E((N_a, N_b), P_a)$ を生成する。そして、C_6 をメッセージ m_6 としてユーザ A に送る。

7. メッセージ m_6 を受信すると、ユーザ A は S_a を用いて C_6 を復号して、その内容を取り出し、受信した N_a を元の値と比較する。値が同じであれば、ユーザ A はユーザ B が許可されたユーザであることを確信する。次に、ユーザ A は鍵 P_b を用いて N_b を暗号化し、暗号文 $C_7 = E(N_b, P_b)$ を生成して、それをメッセージ m_7 としてユーザ B に送る。

8. メッセージ m_7 を受信すると、ユーザ B は S_b を用いて C_7 を復号し、受信した N_b を元の値と比較する。値が同じであれば、ユーザ B もユーザ A が許可されたユーザであることを確認したことになる。相互信頼を確立するにはこれで十分であり、通常の通信を開始することができる。

対称暗号システム及び非対称暗号システムにおける鍵配布に関する上述のプロトコルパラダイムは基本的な設計原則を述べただけである。現実的なプロトコルは、これらの基本的なパラダイムを拡張したものでなければならない。

11.4　認証

認証では、要求された資源へのアクセスを許可する前に、ユーザ (人又はプログラム) の身元を確認する問題を扱う。即ち、認証機構は、要求を行うユーザの身元を確認することによって、許可されていないユーザによるシステム (又はシステムの何らかの資源) の使用を禁止する。

認証は基本的には同定と確認からなる。**同定** (identification) は、ユーザによってある身元が主張されるプロセスであり、**確認** (verification) はユーザが主張した身元を確認するプロセスである。したがって、認証処理の正しさは、使用される確認手続きに大きく依存する。

分散システムで通常必要とされる認証の主なタイプは以下の通りである。

1. **ユーザログイン認証** (user login authentication):ログイン時にユーザの身元がシステムによって確認される。

2. **通信実体の一方向認証** (one-way authentication of communication entities):2 つの通信実体のうち一方によって他方の身元が確認される。

3. **通信実体の双方向認証** (two-way authentication of communication entities):相互認証であり。2 つの通信実体が互いに相手の身元を確認する。

これらのタイプの認証を実行するのに通常用いられる認証機構については以下で述べる。なお、以下で述べる認証プロトコルのパラダイムは設計原則のみを述べたものである。現実的なプロトコルは、これらの基本的なパラダイムを洗練させ、環境に関するより弱い仮定と、より強い事後条件に対処できるものでなければならない。

11.4.1 認証の方法

認証の基本的な方法は以下の通りである [Shankar 1977, Woo and Lam 1992]。

1. **知識による証明** (proof by knowledge):この方法では、認証は許可された当事者のみが知り得るような何かを確認することである。ユーザによって与えられるパスワードに基づいたユーザ認証は、知識による証明の一例である。知識による証明の考え方に基づく認証方式は、さらに2つのタイプに分類される。即ち、直接提示方式と挑戦応答方式である。**直接提示方式** (direct demonstration method) では、ユーザがある種の情報 (例えばパスワード) を与えることによって自分の身元を主張し、確認者はあらかじめ保存されている情報と与えられた情報を比較して、認証を行う。一方、**挑戦応答方式** (challenge-response method) では、ユーザは確認者が与える挑戦的な質問に正しく答えることによって、自分の身元を主張する。例えば、ユーザとしてシステムに参加する時に、ユーザはある関数、例えば x+18 を選ぶ。そのユーザがログインする場合、システムはある1つの数値をランダムに選び、それを表示する。例えば、その数値が 105 であった場合、認証に成功するためには、ユーザは 123 をタイプしなければならない。さらにセキュリティを高めるためには、同じユーザが複数の関数を用いる場合もある。どの関数が用いられるかは、いつログインするかに依存する。例えば、ユーザが7つの関数を用意し、週に各曜日ごとに異なる関数を使用することもできる。この方法の変形として、父親の名前、母親の名前、自宅のある通りの名前などの質問リストをシステムが用意することもできる。ユーザがシステムに参加する時には、これらのすべての質問に答えなければならず、システムはその答えを保存しておく。ログイン時には、システムはこれらの質問の中から1つをランダムに選び、それに対するユーザの答えを検証する。

2. **所有による証明** (proof by possession):この方法では、ユーザは許可された当事者だけが所有しうるアイテムを提示することによって身元を主張する。システムは、提示されたアイテムを検証し、主張された身元を確認するように設計される。例えば、目に見えない電気的な形態でユーザ識別番号が書かれた磁気ストライプを貼付したプラスチックカードを、ユーザが提示するアイテムとして用いることができる。ユーザはシステム端末のスロットにカードを挿入する。システムはカードからユーザ識別番号を抽出し、提示されたカードが許可されたユーザのものであるかどうかを確認する。提示されるアイテムが偽造されておらず、安全に保護されている時にしか、セキュリティが保証されないことは明らかである。

3. **特性による証明** (proof by property):この方法では、偽造困難な、ユーザの物理的特性を計測することによって、そのユーザの身元を確認するように設計される。計測される特性は識別可能なものでなければならない。即ち、システムを使用する可能性があるすべてのユーザに関してユニークでなければならない。例えば、ある特殊装置 (生理計測装置と呼ばれる) がシステムの各端末に接続され、容姿、指紋、手の形状、声、サインなどのユーザの物理的特性を検証する。計測すべき特性を決定するには、その方法がユーザ団体に心理的に受け入れられるよう考慮することが重要である。生理計測システムは、ユーザが主張通りの人物であるかどうかを判定する最も信頼できる方法である。しかし、一般的に最も高価な方法でもある。さらに、生理計測装置はユーザにとって不当に侵害的に見えることから、ユーザに受け入れられない場合も多い。

上述の 3 つの認証方法の中で、知識による証明と所有による証明は、安全な分散システムにおいて必要となるすべてのタイプの認証に適用可能であるが、特性による証明は、特殊計測装置を備えたシステムによる、人間ユーザの認証のみに限定されることが一般的である。さらに、実際には、システムはこれらの認証方法の 2 つ以上を組み合わせて使用することもできる。例えば、自動現金支払い機で用いられる認証機構は、通常、知識による証明と所有による証明を使用している。即ち、ユーザは正しい ID カードを提示し、そのカード上の識別番号に対応したパスワードを入力する場合に限って、現金を引き出すことができる。

11.4.2 ユーザログイン認証

集中型システムと同様に、分散システムでも、ユーザはシステム内のホストにログインすることによって、そのシステムへのアクセス権を獲得する。ユーザの身元はログイン時に確立され、それ以降のすべてのユーザ活動は、この身元に則して行われる。ログイン時の、正しいユーザ身元確認は、システムが安全に機能するために必須である。これは、すべてのアクセス制御とアカウント機能が、この身元に基づいて行われるからである。ユーザログイン認証に関しては、3つの認証方法のどれもが使用可能であるが、知識による証明が最も広く用いられている。特に、大部分のシステムでは、パスワードに基づく直接提示方式が用いられている。

パスワードに基づく認証方式では、システムは許可されたユーザのログイン名とそれに対応するパスワードの表を管理する。ユーザがログインしたい場合、システムはパスワードを入力するよう求める。ユーザが与えたパスワードが、システムが保持している当該ユーザのパスワードと一致した場合に、そのユーザは合法的とみなされ、ログインが許される。一致しない場合には、ログイン要求は拒絶される。

優れたセキュリティを提供し、かつ実用的であるためには、パスワードに基づく認証システムは以下のような機構を備えなければならない。

1. パスワードを秘密に保つこと
2. 類推しにくいパスワードとすること
3. 破られたパスワードによる被害を限定できること
4. 許可されていないユーザのログインを特定でき、妨げられること
5. システムの全資源の使用が、1回のログインでできること

以下では、これらの問題を扱うためによく用いられる機構を説明する。

パスワードの秘密性維持

コンピュータシステム外(外界)では、ユーザがパスワードの機密性に責任を持つ。外界においてパスワードの秘密性をいかに保つかは、オペレーティングシステム設計者の関知するところではない。唯一の例外は、パスワードが入力されたときに、端末近辺の人によるのぞき見を避けるために、そのパスワードを端末の表示画面上に表示しないことである。主な関心事は、侵入者がシステ

ムのパスワード表にアクセスして、誰かのパスワードを獲得するのを防ぐことである。もちろん、パスワード表は保護されており、許可されたプログラムだけがアクセス可能である。しかし、パスワード表が偶然に明らかにされたり、システム管理者がその表にアクセスすることは大いに起こりうることである。したがって、 名前とパスワードは、平文の形で格納する代わりに、暗号化され、暗号文の形で表に格納される。この場合、ユーザによって指定された名前とパスワードを表検索に直接使用する代わりに、それらはまず暗号化され、その結果が表検索に用いられる。この方式の実装に関する主な要求は、暗号化関数とパスワード表の両方を知ったとしても、元のパスワードを知ることを不可能にすることである。即ち、この方式では暗号文を復号する必要がないので、暗号化には非可逆関数が必要である。これは**一方向暗号** (one-way cipher) の使用例である。暗号システムでは、以下のような関数を用いる。

$$C = E(P, K)$$

ここで、K は鍵、P は平文、C は結果として得られる暗号文である。一方向暗号は、以下のように、平文 P を E の鍵とすることによって実装される。

$$C = E(P, P)$$

一方向暗号では、たとえ侵入者が鍵を知ったとしても、暗号化関数 (E) の逆関数を実行することが極めて難しい。さらに、鍵が未知の場合 (パスワード自体が鍵として用いられるよなパスワードセキュリティ方式の場合にように) には、逆関数の実行はさらに難しくなる。それは、侵入者はどの関数が用いられているか (及び、その関数を何回用いるか) を知ることができないからである。

分散システムにおいてパスワードの秘密を保つためには、パスワードが平文の形でネットワークを通して送られることがないようにすることが重要である。さらに、パスワードを通常のサーバに格納してはならず、十分に保護された、信頼できるサーバのみに格納すべきである。このような要求のため、許可を求めるためにユーザが単にパスワードを認証サーバに送るだけといった認証方式は、分散環境ではうまく働かない。ケルベロス認証システムはこの問題に対して、興味ある解を提供している。これに関しては本章の後半で述べる。

パスワード推測の困難化

パスワードの秘密性を保つ機構を用いても、システムのセキュリティが破られないことは保証できない。秘密性の確保は、パスワードを得ることを困難にしているだけである。侵入者は常に試行錯誤方式を使うことが可能である。パスワードの場合には、パスワードと名前の組を類推することで破られるのが普通である。侵入者がどれだけうまくパスワードを推測できるかは、MorrisとThompson[1979]によるUNIXシステムのパスワードに関する研究から明らかである。この研究では、まず、人の姓名、都市や通りの名前、小さな辞書から選ばれた単語やその逆の綴り、ランダムに選ばれた文字から成る短い文字列、車のプレートナンバーなどを含む、ありそうなパスワードのリストを作成する。次に、これらのパスワードを、既知の暗号化アルゴリズムで暗号化し、暗号化された推測パスワードリストを得る。また、各種UNIXシステムのパスワード表のエントリを用いて、もう1つの暗号化パスワードリストを作成する。これら2つのリストのエントリを比較して、一致するものを見つけ出す。驚いたことに、実際のパスワードの86%以上が、推測パスワードリストに一致するものを持つことが分かった。

限られた数のありそうな文字列をテストするだけでも、大部分のパスワードを明らかにできてしまう傾向がある。これは、覚えやすいように比較的短く単純なパスワードが選ばれる傾向があるからである。パスワードの推測を困難にするために用いられる方法には以下のようなものがある。

1. **長いパスワード** (longer password):パスワードの長さが、網羅的手法によるパスワードの発見の容易さを決定する。例えば、3桁のパスワードは、1000種類あるが、4桁になると10000種類もある。パスワードが長ければ、数え上げの影響を受けにくくなる。すべての可能なパスワードを列挙する仕事が、パスワードの長さが長くなるに伴って増大するからである。長いパスワード使用は、パスワード登録プログラムによって強制したり、推奨したりすることができる。同プログラムは、ユーザが短いパスワードを入力した場合に、より長いものに変更するよう要求することができる。

2. **パスワード表粉飾** (salting password table):パスワードの推測を困難にするもう1つの方法は、各パスワードが最初に入力された時に、nビットの乱数を付加して、人工的にパスワードの長さを長くすることである。パスワードが変更された時には、その乱数も変更される。単に暗号化されたパスワードをパスワード表に格納する代わりに、まずパスワードと乱数が連結され、それらが

一緒に暗号化される。この暗号化結果が、パスワード表に格納される。この方式では、パスワード表は暗号化されていない形式の乱数を格納するための付加的な欄を持つ。ログイン時には、この乱数が入力されたパスワードに連結され、暗号化され、そして格納されている暗号化値と比較される。もし侵入者があるユーザのパスワードを Tokyo と考えたとしても、単に Tokyo を暗号化し、その結果をパスワード表に格納されている値と比較するだけでは、もはや不十分であることから、パスワードの推測は困難となる。推測を行うたびに、侵入者は 2^n 個の文字列を試さなければならない。例えば、$n = 4$ の場合には、Tokyo0000、Tokyo0001、Tokyo0002 などである。これによって、各推測に要する暗号化時間と比較時間が増大する。UNIX では、この方式が用いられており、$n = 12$ である。

3. **パスワード選択におけるシステム補助** (system assistance in password selection):パスワードはシステムが生成してもよいし、ユーザが選択してもよい。ユーザ選択のパスワードは、多くの場合容易に推測できる。ユーザが推測困難なパスワードを選択することを支援するようにシステムを設計することができる。これは 2 つの方法によって実現される。1 つの方法は、容易に推測可能なパスワードのリストをシステムに格納しておき、まずユーザ選択パスワードと、そのリストのエントリを比較する。一致するものが発見された場合には、システムは選択されたパスワードを拒絶し、ユーザには、それが容易に推測できることを通知して、他のパスワードを選択するように要求する。システム支援を提供するもう 1 つの方法は、mounce、bulbul、halchal のように、パスワードをして用いることができる、ランダムで、覚え易く、無意味な単語を生成するパスワード生成プログラムを用意することである。ユーザはシステムによって勧められたパスワードを選択し、その単語の文字を好みに応じて大文字にしたり、小文字にして、自分のパスワードとする。例えば、hAlCHal のようにである。

許可されていないユーザログインの特定と妨害

許可されていないユーザログインに対するシステムセキュリティを向上させるには、幾つかの管理手法を使用すべきである。以下では 3 つの手法を紹介する。

1. **脅威監視** (threat monitoring):この手法は、疑わしい活動パターンの確認

によって、セキュリティ違反を検出する。例えば、ユーザログインの際に、誤ったパスワードを入力した回数を数えておき、例えば3回の失敗があった後には、システムは警報をセットして、セキュリティ担当者に通知する。

2. **監査ログ** (audit log):この手法では、システムはすべてのログ記録を維持しておく。即ち、ログインが行われた時刻、ログイン所要時間、アクセスされたオブジェクトとそのアクセスタイプなどが各ログインに関して記録される。ユーザがログインに成功した時、システムはユーザに対して、前回のログインに関して記録されている情報(時刻や所要時間など)を報告する。このことは、ユーザが不法侵入を検出するのに役に立つ。セキュリティ違反が検出された後には、監査ログを用いて、侵入者によってアクセスされたオブジェクトを検出し、損害の度合いを知ることができる。この情報は、侵入者によってもたらされた損害から回復する上で役に立つ。

3. **餌付きの罠** (baited trap):餌付きの罠によって侵入者を捕獲することもできる。例えば、システムが簡単なパスワードを持つ特別のログイン名を用意しておく。ログイン名としてuser、そのパスワードとしてuserを用いるなどである。これらの名前を用いて誰かがログインした場合には、即座にシステムセキュリティ担当者に通知される。

単一のログイン

分散クライアントサーバ環境では、ユーザは自分のノード上で複数のクライアントプログラムを走行させ、これらは遠隔ノード上の複数のサーバプログラムにアクセスする。このような環境では、サーバはクライアントが正当なユーザのために走行していることを認証しなければならない。もし単純なパスワード準拠の認証方式が用いられると、そのユーザのために走行している各クライアントプログラムを認証しようとするたびに、パスワードが提示されなければならない。これは煩わしく不便辺である。ユーザは、新しいサービスにアクセスするたびにパスワードを入力したいとは思わないからである。また、このことは透過性の観点からも望ましいことではない。透過性は、仮想的な単一システムイメージを提供することを目的としているからである。この問題を解決する簡単な方法は、パスワードをユーザホストにキャッシュしておき、新しいサービスをアクセスする時には、そのキャッシュからパスワードを取り出して使用することである。しかし、パスワードを秘密に保つという観点からは、パ

スワードをキャッシュに保持しておくことは危険であるため、この解決策はうまく機能しない。これに関しても、ケルベロス認証システムが興味ある解決策を提供しており、それについては本章の後半で述べる。

破られたパスワードによる被害の限定

パスワードを頻繁に変更することが推奨されている。その結果、侵入者が現在のパスワードの推測に成功しても、ユーザがそれを変更した後には、新たに推測を行う必要が生じる。この方法では、侵入者によってもたらされる被害が軽減できる。この方法の極端な形態が**ワンタイムパスワード** (one-time password) である。ワンタイムパスワードの考え方を実装するには、3 つの異なる方法がある。最初の方法では、ユーザはパスワードリストが載った本を得る。各ログインにおいて、リスト内の次のパスワードを用いなければならない。パスワード資料を安全な場所に保管しなければならないことは明らかである。2 番目の方法は、ユーザがログアウトするたびに、次のログインのためのパスワードを選ぶようにシステムが要求し、古いパスワードを新しいもので置き換える。ユーザはその新しいパスワードを覚えておくかメモを取っておくことによって、次回のログイン時の使用に備える。3 番目の方法は、スマートカードや同期型パスワード生成器などの特殊装置に依存する。例えば、同期型パスワード生成器は、擬似乱数的な英数字からなる単語を発生するもので、その値は例えば 1 分おきに変更されると同時に、コンピュータ内のデータベースと時間的に同期している。ログインするためには、ユーザはログイン時にカードに表示された単語を入力する。その結果、その時点のみで有効で、しかも一回しか使えないワンタイムパスワードを得ることができる。このような装置の 1 つとして、マサチューセッツ州ケンブリッジ市の Security Dynamics 社が開発した SecureID がある。今までのところ、この方法は必ずしも成功を収めていない。その主な原因は、利便性の悪さと、付加的なハードウェアによるコストにある。即ち、すべてのユーザがハードウェアデバイスを購入しなければならない。しかし、一旦設置すれば、管理が極めて容易であるという利点がある (その反対に、マニュアルパスワードリストでは、全ユーザを頻繁に更新する必要がある)。

11.4.3 通信実体間の一方向認証

ある実体Aがもう一方の実体Bと通信をしたい時、BはAに通信を許可する前に、その身元を確認する必要がある。例えば、サーバは通信を求めてくる任意のクライアントの身元を確認するように設計される。以下では、通信実体間の一方向認証によく用いられるプロトコルについて述べる。

以下の説明は [Woo and Lam 1992] に述べられた内容に基づいている。認証プロトコルのパラダイムは直接的に暗号システムを用いるので、その設計原則は用いられる暗号システムのタイプに密接に関連している。したがって、プロトコルは大きく分けて2つの範疇に分類される。即ち、対称暗号システムに基づくものと、非対称暗号システムに基づくものである。両範疇の認証プロトコルは共に、知識による証明原則に基づいている。

対称暗号システムに基づくプロトコル

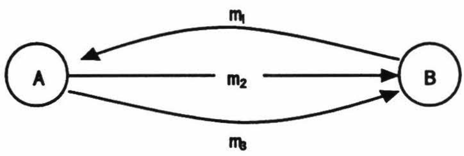
対称暗号システムでは、共有鍵に関する知識によって、通信実体は任意のメッセージを暗号化したり、復号したりできる。このような知識がなければ、メッセージを暗号化したり、暗号化されたメッセージを復号することができない。したがって、対称暗号システムに基づく認証プロトコルでは、確認者は、要求者の身元に合致した実体(確認過程で用いられる実体以外)のみが知っていると信じる鍵を用いて要求者が正しくメッセージを暗号化したかどうかを確認することによって、要求者の身元を確認する。

ユーザAがBと通信したいと思っているが、Bは通信を始める前にAを認証しようとしていると仮定する。また、Kは対称暗号システムの鍵であり、ユーザAとBで共有されているものとする。この場合の認証過程は以下のステップから構成される。

1. ユーザAは自己の識別子 (ID_a) を、鍵 K を用いて暗号化し、暗号文 $C_1 = E(ID_a, K)$ を得る。次に、メッセージ m_1 をユーザBに送るが、それには ID_a と C_1 が含まれる。

2. メッセージ m_1 を受け取ると、ユーザBは鍵 K を用いて C_1 を復号し、得られた結果をメッセージの ID_a と比較する。一致すれば、ユーザAは受け入れられる。一致しなければ、拒絶される。

このプロトコルの主な弱点は、再送に弱いという点である。即ち、メッセージ m_1 を記録しておき、後でBに対してそれを再送することによって、侵入者がAになりすますことができる。再送攻撃に対しては、ノンス又はタイムスタンプを用いて対処することができる。メッセージの再送問題を克服する、ノンスに基づいた挑戦応答プロトコルは以下のように動作する (図11.6を参照)。



m1 = IDa = ユーザAの識別子
m2 = Nr = ユーザBによって生成された乱数
m3 = C1 = E(Nr, K)
ここで、Kはユーザ A と B で共有される対称鍵

BはKを用いてC1を復号し、その結果を元のNrと比較する。一致していれば、Aは受け入れられるが、そうでなければ拒絶される。

図 11.6　対称暗号システムに基づく一方向認証プロトコル

1. ユーザAは識別子 (ID_a) を平文形式でメッセージ m_1 に入れて、ユーザBに送る。

2. メッセージ m_1 を受け取ると、ユーザBは乱数 N_r を生成し、それを平文形式でメッセージ m_2 に入れてユーザAに送る。

3. メッセージ m_2 を受け取ると、ユーザAは鍵 K を用いて N_r を暗号化し、暗号文 $C_1 = E(N_r, \mathrm{K})$ を得る。次に、メッセージ m_3 として、C_1 をBに送る。

4. メッセージ m_3 を受け取ると、ユーザBは鍵 K を用いて C_1 を復号し、その結果を元の N_r と比較する。一致すれば、ユーザAは受け入れられるが、一致しなければ拒絶される。

このプロトコルでは、N_r の新鮮さ (その値は通信のたびに異なる) が、AとBの間の認証された古い通信の記録を再送することによって、侵入者がAにな

りすますことができないことを保証する。

上述のプロトコル機能は正しく機能するが、以下の理由から一般の大規模システムへの適用は現実的ではない。

1. この方式では、各ユーザが認証したいと思う他のすべてのユーザの秘密鍵を保持しておかなければならない。余りにも多くのユーザを有し、またそのユーザも頻繁に変更されるような大規模システムでは、これはほとんど実現不可能である。

2. 一人のユーザが破られることによって、システム全体が破られることが起こりうる。

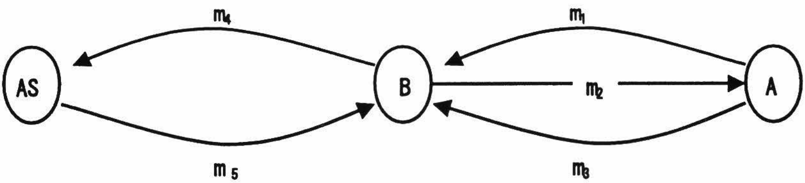
これらの問題を克服するために、集中型の**認証サーバ** (authentication server)(AS) を用いる方式が [Needham and Schroeder 1978] によって提案されている。システム内の各ユーザは AS とあらかじめ用意された秘密鍵を共有する。AS は一般的に信頼できる実体であり、システム内で通信を行うすべてのユーザによって共有される。AS が用いられる場合には、認証プロトコルは図 11.7 に示すような形となる (ここで、K_a と K_b はそれぞれユーザ A と B の秘密鍵であり、これらが AS と共有されていると仮定している)。

1. ユーザ A は識別子 (ID_a) を平文形式でメッセージ m_1 に入れて、ユーザ B に送る。

2. メッセージ m_1 を受け取ると、ユーザ B は乱数 N_r を発生し、これを平文形式でメッセージ m_2 に入れて、ユーザ B に送る。

3. メッセージ m_2 を受け取ると、ユーザ A は秘密鍵 K_a を用いて N_r を暗号化し、暗号文 $C_1 = E(N_r, K_a)$ を得る。これをメッセージ m_3 としてユーザ B に送る。

4. メッセージ m_3 を受け取ると、ユーザ B は秘密鍵 K_b を用いて (ID_a, C_1) を暗号化し、暗号文 $C_2 = E((ID_a, C_1), K_b)$ を生成する。これをメッセージ m_4 として、AS に送る。



AS = 認証サーバ

IDa = ユーザ A の識別子

IDb = ユーザ B の識別子

Ka = ユーザ A の秘密鍵

Kb = ユーザ B の秘密鍵

ID_a	K_a
ID_b	K_b
:	:

各ユーザの秘密鍵表

m_1 = IDa

m_2 = Nr = ユーザ B によって生成された乱数

m_3 = C_1 = E(Nr, Ka)

m_4 = C_2 = E((IDa, C_1), Kb)

AS は Kb で C_2 を復号することによって Nr を検索し、次に Ka で C_1 を復号する。

M_5 = C_3 = E(Nr, Kb)

B は Kb を用いて C_3 を復号し、その結果を元の Nr と比較する。一致していれば、A は受け入れられるが、そうでなければ拒絶される。

図 11.7　対称暗号システムに基づきかつ集中型認証サーバを用いた一方向認証プロトコル

5. メッセージ m_4 を受け取ると、ASは鍵 K_b で C_2 を復号し、(ID_a, C_1) を抽出する。そして、ASは自身のデータベースから ID_a に対応する鍵 K_a を検索し、その鍵で C_1 を復号して N_r を抽出する。次に、鍵 K_b で N_r を暗号化し、暗号文 $C_3 = E(N_r, K_b)$ を生成して、これをメッセージ m_5 としてユーザBに送る。

6. メッセージ m_5 を受け取ると、ユーザBは秘密鍵 K_b でそれを復号して N_r を抽出し、元の N_r と比較する。一致すれば、ユーザAは受け入れられ、一致しなければ拒絶される。

前に述べたプロトコルと比較して、鍵の配布と保持の問題が大きく軽減されている。これは、各ユーザが1つの鍵だけを管理すれば良いからである。さらに、破られることの危険がほとんどASに移行しているので、単にASのセキュリティを強固にすることによって、システムのセキュリティが大幅に向上する。しかし、集中型のASは集中型のKDCと同じ欠点を持っている。これらの欠点も、完全なまたは部分的な分散型ASを用いることによって解決されうる。これは、11.3.4の鍵配布問題で述べたものと同様の方法による。

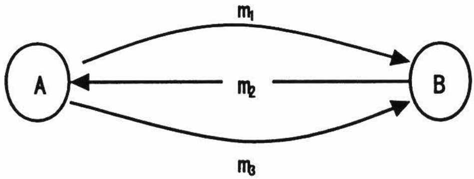
非対称暗号システムに基づくプロトコル

非対称暗号システムでは、各ユーザの公開鍵は公になっているが、各ユーザの秘密鍵はそのユーザだけが知っており、他の誰も知らない。したがって、非対称暗号システムに基づく認証プロトコルでは、確認者は、身元を主張するユーザの秘密鍵を用いて要求者がメッセージを正しく暗号化できているかどうかを確認することによって、要求者の身元を確認する。

ユーザAがBと通信したいと思っているが、Bは通信を始める前にAを認証しようとしていると仮定する。また、P_a と S_a をユーザAの公開鍵、秘密鍵とし、P_b と S_b をユーザBの公開鍵、秘密鍵とする。この場合の認証過程は以下のステップから構成される(図11.8を参照)。

1. ユーザAは識別子 (ID_a) を平文形式でメッセージ m_1 に入れてユーザBに送る。

2. メッセージ m_1 を受け取ると、ユーザBは乱数 N_r を生成し、それを平文形式でメッセージ m_2 に入れてユーザAに送る。



m₁ = IDa = ユーザAの識別子

m_1 = IDa = ユーザAの識別子

m_2 = Nr = ユーザBによって生成された乱数

m_3 = C_1 = E(Nr, Sa)

ここで、Sa = ユーザAの秘密鍵

Bはユーザ A の公開鍵を用いて C_1 を復号し、その結果を元のNr と比較する。一致していれば、A は受け入れられるが、そうでなければ拒絶される。

図 11.8　非対称暗号システムに基づく双方向認証プロトコル

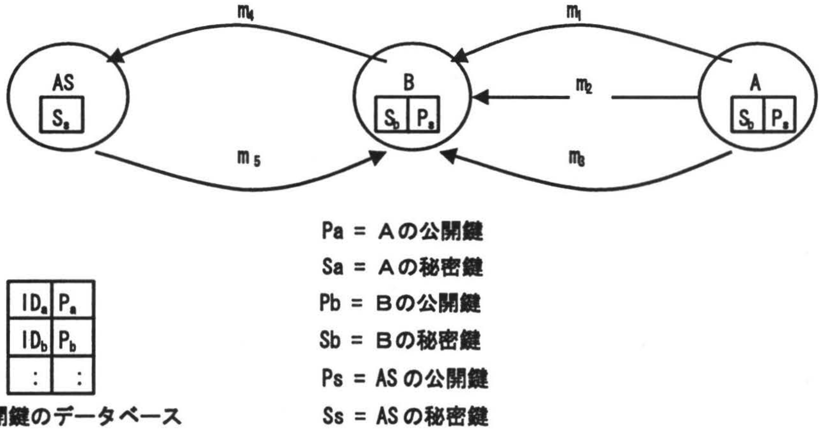
3. メッセージ m_2 を受け取ると、ユーザ A は秘密鍵 S_a を用いて N_r を暗号化し、暗号文 $C_1 = E(N_r, S_a)$ を得る。それをメッセージ m_3 としてユーザ B に送る。

4. メッセージ m_3 を受け取ると、ユーザ B はユーザ A の公開鍵 P_a を用いて C_1 を復号し、得られた結果を元の N_r と比較する。一致していれば、ユーザ A は受け入れられるが、一致していなければ拒絶される。

対称暗号システムに基づくプロトコルの場合と同様に、この場合も、鍵の配布と保持の問題を軽減するために集中型 AS を用いることができる。この場合には、AS はすべての公開鍵を保持するデータベースを維持し、システムの各ユーザは AS の公開鍵 P_s のコピーを管理する。AS が使用される時には、認証プロトコルは以下のような形となる (図 11.9 を参照)。

1. ユーザ A は識別子 (ID_a) を平文形式でメッセージ m_1 に入れてユーザ B に送る。

2. メッセージ m_1 を受け取ると、ユーザ B は乱数 N_r を生成し、それを平文



Pa = Aの公開鍵
Sa = Aの秘密鍵
Pb = Bの公開鍵
Sb = Bの秘密鍵
Ps = AS の公開鍵
Ss = AS の秘密鍵
IDa = Aの識別子
IDb = Bの識別子

m_1 = IDa

m_2 = Nr = Bによって生成された乱数

m_3 = C_1 = E(Nr, Sa)

m_4 = (Rb, IDa)

　　ここで、Rb = ユーザの公開鍵を要求するコード

m_5 = C_2 = E((IDa, Pa), Ss)

BはPs で C2 を復号することによってNr を検索し、次にPa で C1 を復号する。このようにして得られたNr の値は、元のNr と比較される。一致していれば、A は受け入れられるが、そうでなければ拒絶される。

図 11.9　非対称暗号システムに基づきかつ集中型認証サーバを用いた一方向認証プロトコル

形式でメッセージ m_2 に入れてユーザ A に送る。

3. メッセージ m_2 を受け取ると、ユーザ A は秘密鍵 Sa を用いて N_r を暗号化し、暗号文 $C_1 = E(N_r, S_a)$ を得る。それをメッセージ m_3 としてユーザ B に送る。

4. メッセージ m_3 を受け取ると、ユーザ B は (R_b, ID_a) を平文形式でメッセージ m_4 に入れて AS に送る。ここで、R_b は、識別子がメッセージの 2 番目の要素として指定されているユーザの公開鍵を要求するための要求コードである。

5. メッセージ m_4 を受け取ると、AS は識別子が ID_a であるようなユーザの公開鍵 P_a をデータベースから検索する。次に、(ID_a, P_a) を自身の秘密鍵 S_s で暗号化し、暗号文 $C_2 = E((ID_a, P_a), S_s)$ を生成する。最後に、それをメッセージ m_5 としてユーザ B に送る。

6, メッセージ m_5 を受け取ると、ユーザ B は AS の公開鍵 P_a を用いて C_2 を復号し、(ID_a, P_a) を抽出する。次に、鍵 P_a を用いて C_1 を復号し、その結果を元の N_r と比較する。一致していれば、ユーザ A は受け入れられ、一致していなければ拒絶される。

この場合、各ユーザは 1 つの公開鍵、即ち AS の公開鍵のみを管理すればよいことに注意する必要がある。また、上述のプロトコルでは、非対称暗号システムが可換であることを仮定していることにも注意する必要がある。即ち、公開鍵と秘密鍵はいずれの順序で適用しても有効である。したがって、公開鍵が暗号化に用いられた場合には、秘密鍵をその復号に用いることができる。一方、秘密鍵が暗号化に用いられた場合には、公開鍵をその復号に用いることができる。

11.4.4　通信実体間の双方向認証

分散システムでは、集中型システムよりも高いスループットや資源利用の平準化を達成するために、仕事を複数のホストに分散させるのが普通である。このような分散された仕事の正しさは、その仕事に参加している仲間プロセスが

お互いを正しく特定できるかどうかに依存している。双方向認証プロトコルによって、両方の通信実体がお互いの身元を確認でき、両者間に安全な論理通信チャネルを確立することができる。

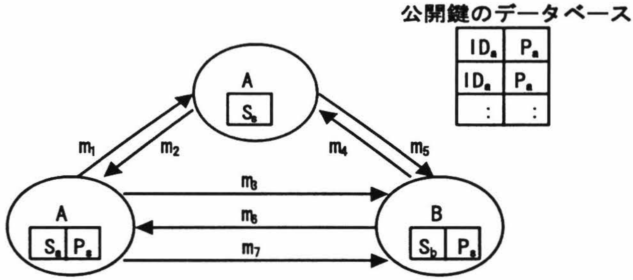
一方向認証を2回行うことで相互の認証が達成できることは明らかである。即ち、2つの通信ユーザAとBがお互いを認証したい場合には、まずAが一方向認証によってBを認証し、次に、AとBの役割を逆転させて同じ処理過程を繰り返すことで、BがAを認証することができる。しかし、この方法は双方向認証用に設計されたプロトコルに比べて高価であることが分かる。例えば、双方向認証を行うために図11.9のプロトコルを2回繰り返す場合、全体で10個のメッセージが必要となる。一方、7個のメッセージだけで済むような相互認証向けプロトコルが以下のように記述できる。これは、非対称暗号システムに基づいたものであり、集中型のASを用いている。以下の記述では、2つのユーザAとBがお互いを認証したいとする。ここで、P_a と S_a はユーザAの公開鍵、秘密鍵であり、P_b と S_b はユーザBの公開鍵と秘密鍵である。さらに、P_s と S_s はASの公開鍵、秘密鍵である。認証プロトコルは以下のステップから構成される(図11.10を参照)。

1. ユーザAが、要求メッセージ (m_1) をASに送り、ユーザBとの間で安全な論理通信チャネルを確立したい旨を表明する。そのメッセージは要求コード (R_a)、ユーザAの識別子 (ID_a) 及びユーザBの識別子 (ID_b) を含む。このメッセージは平文形式で送られる。

2. メッセージ m_1 を受け取ると、ASはメッセージ内の ID_b に対応する公開鍵 P_b をデータベースから検索する。自身の秘密鍵 S_s を用いて、ASは (ID_b, P_b) を暗号化し、暗号文 $C_1 = E((ID_b, P_b), S_s)$ を生成する。これをメッセージ m_2 としてAに送る。

3. メッセージ m_2 を受け取ると、ユーザAはASの公開鍵 P_s を用いて C_1 を復号し、その内容を抽出する。次に、乱数 N_a を生成し、(ID_a, N_a) をユーザBの公開鍵 P_b を用いて暗号化して、暗号文 $C_2 = E((ID_a, N_a), P_b)$ を生成する。これをメッセージ m_3 としてBに送る。

4. メッセージ m_3 を受け取ると、ユーザBは秘密鍵 S_b を用いて C_2 を復号し、その内容を抽出する。次に、メッセージ m_4 をASに送り、ユーザAの公



Ps = AS の公開鍵　　Pb = ユーザ B の公開鍵
Ss = AS の秘密鍵　　Sb = ユーザ B の秘密鍵
Pa = ユーザ A の公開鍵　　IDa = A のユーザ識別子
Sa = ユーザ A の秘密鍵　　IDb = B のユーザ識別子

m_1 = (Ra, IDa, IDb)

　　ここで、Ra = ユーザ A によってなされた要求のコード

m_2 = C_1 = E((IDb, Pb), Ss)

m_3 = C_2 = E((IDa, Na), Pb)

　　ここで、Na はユーザ A によって生成された乱数

m_4 = C_3 = E((Rb, IDa, IDb, Na), Ps)

m_5 = (C_4, C_6)

　　ここで、C_4 = E((IDa, Pa), Ss)

　　　　C_6 = E(C_5, Pb)

　　　　　　ここで、C5 = E(IDb, K, N, Ss)

　　　　　　　　ここで、K はユーザ A と B のために AS が生成するセッション鍵

m_6 = C_7 = E((C_5, Nb), Pa)

　　ここで、Nb はユーザ B によって生成された乱数

m_7 = C_8 = E(Nb, K)

B はセッション鍵 K で C_8 を復号し、その結果を元の Nb と比較する。一致していれば、A と B の間に鍵 K を持って確立された論理通信チャネルは、新しく確立されたものであり、安全であることを証明するに十分である。

図 11.10　非対称暗号システムに基づきかつ集中型認証サーバを用いた双方向認証プロトコル

開鍵と、AとBの間の安全な論理通信チャネル用のセッション鍵を要求する。メッセージは要求コード R_b、ID_a、ID_b および N_a を含む。送信前に、このメッセージはASの公開鍵 P_s を用いて暗号化される。即ち、メッセージは暗号文 $C_3 = E((R_b, ID_a, ID_b, N_a), P_s)$ を含む。

5. メッセージ m_4 を受け取ると、ASは秘密鍵 S_s を用いて C_3 を復号し、その内容を抽出する。ASはAとB用の新しいセッション鍵Kを生成する。次に、以下の3つの暗号文を生成する。$C_4 = E((ID_a, P_a), S_s)$、$C_5 = E((ID_b, \mathrm{K}, N_a), S_s)$ 及び $C_6 = E(C_5, S_b)$ である。最後に、C_4 と C_6 をメッセージ m_5 に入れてBに送る。

6. メッセージ m_5 を受け取ると、ユーザBは P_s と S_b を用いて、それぞれ C_4 と C_6 を復号し、それらの内容を抽出する。次に、乱数 N_b を生成し、暗号文 $C_7 = E((C_5, N_b), P_a)$ を作成する。最後に、C_7 をメッセージ m_7 に入れて、Aに送る。ここで、C_5 は C_6 を復号して得られることに注意する。

7. メッセージ m_6 を受け取ると、ユーザAは、まず秘密鍵 S_a を用いて C_7 を復号し、次に、ASの公開鍵 P_s を用いて C_5 を復号する。その結果、AとBの両方がセッション鍵 K を持つことになる。次に、ユーザAは暗号文 $C_8 = E(N_b, \mathrm{K})$ を生成し、それをメッセージ m_7 に入れてBに送る。

8. メッセージ m_7 を受け取ると、ユーザBはセッション鍵Kを用いて C_8 を復号し、その結果と元の N_b を比較する。一致していれば、AとBの間に鍵Kで確立された論理通信チャネルは新しく確立されたものであり、かつ安全であることが証明されるに十分である。ユーザAとBは共に、お互いの身元に確信を持っている。

上述の認証プロトコルは非対称暗号システムに基づいているが、安全な論理通信チャネルが確立した後の、ユーザA、B間の実際の通信は、対称暗号システムによって行われることに注意する必要がある。

11.4.5 事例研究： ケルベロス認証システム

分散システムでは安全な認証が必要であるため、認証標準とそのためのシステムの設計が行われてきた。例えば、X.509はCCITT X.500ディレクトリ認証フレームワークの標準を規定 [Smart 1994] し、ケルベロスはMITで開発さ

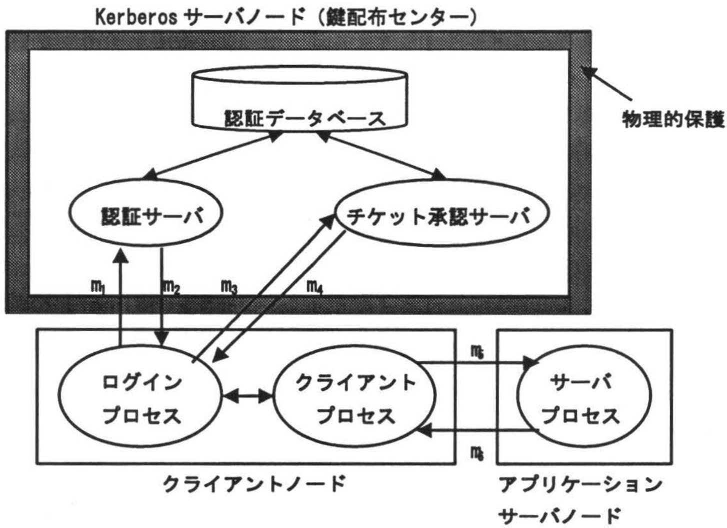
れたネットワーク認証システムである [Neuman and Theodore 1994、Stallings 1994]。また、SPX は Digital Equipment Corporation で開発された実験的な認証システムである [Khanna 1994]。今日までに開発された認証システムの中で、ケルベロスシステムが最も有名であり、進化し続けているものである。インターネット標準であると言われており、ネットワーク化されたクライアントサーバ環境における遠隔認証の業界標準となっている。したがって、本書ではこれについて詳述する。

ケルベロスは MIT の Athena プロジェクト [Champine et al. 1990] の一環として開発された。その名前は、ギリシャ神話において地獄への門を守る3つ頭の犬に因んでいる。その設計は、鍵配布と認証に関する Needham と Schroeder[1978] のアイデアに基づいており、このアイデアに関しては、それぞれ本書の 11.3.4 と 11.4.3 で述べた。現在では、商用の実装とパブリックドメインの実装の両方が利用可能であり、分散環境における安全な資源アクセスを提供するために、MIT をはじめとする各所で広く使用されている。Transarc の AFS ファイルシステムでも用いられているし、OSF の DCE セキュリティサーバ (本章の後半で述べる) の基盤要素ともなっている。DEC、HP、IBM などのメーカも、商用 UNIX や中規模サーバプラットフォームの標準ミドルウェアの一部としてケルベロスを提供している。ここでは、ケルベロスの最新版である、第5版について説明する。

ケルベロスシステムアーキテクチャ

ケルベロスのシステムアーキテクチャを図 11.11 に示す。システムは以下の基本要素から構成される。

1. **ケルベロスサーバ** (Kerberos server):ケルベロスシステムの基本要素はケルベロスサーバであり、それは鍵配布センターの役割を果たす。各ケルベロスサーバは認証データベース、認証サーバ及びチケット承認サーバを持つ。認証データベースは、システムのすべてのユーザの ID とパスワードを保持する。さらに、ケルベロスサーバは唯一の秘密鍵をシステム内の各サーバと共有する。したがって、認証データベースもシステム内のすべてのサーバのサーバ ID と秘密鍵を持つ。パスワードと秘密鍵は、ケルベロス実装の一部として、物理的な方法を含む何らかの安全な方法で分散させられている。ケルベロスは鍵の生成とメッセージの暗号化に DES アルゴリズムを用いているが、これは別のモジュールとして実装されており、他の適当なアルゴリズムに容易に置き換え



IDg = チケット承認サーバの識別子
IDc = クライアントの識別子
IDs = アプリケーションサーバの識別子
Ni = ナンス
Kc = クライアントの秘密鍵
Ks = アプリケーションサーバの秘密鍵

Kg = チケット承認サーバの秘密鍵
K_1 = チケット承認チケットセッション鍵
K_2 = サービス承認チケットセッション鍵
Tsi = チケット有効期間の開始時刻
Tei = チケット有効期間の終了時刻
Ti = タイムスタンプ

$m_1 = (ID_c,\ N_1)$
$m_2 = C_2 = E(N_1,\ K_1,\ C_1),\ K_c)$，ここで $C_1 = E((ID_C,\ ID_g,\ T_{s1},\ T_{e1},\ K_1),\ K_g)$
$m_3 = (ID_s,\ N_2,\ C_1,\ C_3)$，ここで $C_3 = E((ID_C,\ T_1),\ K_1)$
$m_4 = C_5 = E((N_2,\ K_2,\ C_4),\ K_1)$，ここで $C_4 = E((ID_C,\ ID_s,\ T_{s2},\ T_{e2},\ K_2),\ K_s)$
$m_5 = (C_4,\ C_6)$，ここで $C_6 = E((ID_C,\ T_2),\ K_2)$
$m_6 = C_7 = E(T_3,\ K_2)$，ここで $T_3 = T_2+1$

図 11.11　Kerberos 認証プロトコル

ることができる。

認証サーバは、ログイン時にユーザの身元を確認するという仕事を行うが、その時、パスワードをネットワークを介して送付することを必要としない。ケルベロスは単一ログイン機能を持つ。したがって、ユーザがログイン後にアクセスする資源の数に無関係に、パスワードを入力するのは 1 回だけである。

チケット承認サーバは、システム内のサーバへのアクセス許可を求めるクライアントに対して、チケットを供給するという仕事を行う。これらのチケットは、相互認証によってクライアントとサーバの間に安全な論理通信チャネルを確立するために用いられる。

ケルベロスは、秘密に保たなければならない貴重な情報を認証データベース内に持つため、これを十分に保護され物理的に安全なマシンに実装することが極めて重要である。ケルベロスサーバを他のアプリケーションと同じマシンに実装することに技術的な問題はないが、セキュリティの観点からは、ケルベロスサーバ専用のマシンを用意することが望ましい。このマシンへのアクセス権を持つユーザの数は極端に制限するべきである。

2. **クライアント** (client):ケルベロスシステムの 2 番目の構成要素は、クライアントプロセスから構成される。通常、クライアントプロセスは実質上公開の場に置かれたワークステーションの上で走行する。公開の場に置かれていることから、そのコンソールは、偶然その前にいたユーザにとっても使用可能である。したがって、クライアントプロセスは全く信頼できないものである。ユーザ (クライアントプロセスは、このユーザのために実行される) は、システム内の他のサーバにアクセスを試みる前に、まずケルベロスサーバに身元を確認してもらわなければならない。一旦ユーザの身元が確認されれば、そのユーザのために実行されるクライアントプロセスは、アクセスしたいサーバとの通信のためのチケットをチケット承認サーバから獲得しなければならない。

3. **アプリケーションサーバ** (application server):ケルベロスシステムの 3 番目の構成要素はアプリケーションサーバである。これは、単にサーバと呼ばれることも多い。あるサーバは、クライアントの信憑性を確認した後にだけ、要求に応じてクライアントに対して特定のサービスを提供する。通常、サーバは適度に安全な部屋に設置されたマシン上で走行する。したがって、ケルベロスは 1 つのサーバを破ることが、他のサーバを破ることにはならないことを保証している。

ケルベロス認証プロトコル

基本構成要素とその機能に関する考え方を用いて、ケルベロスプロトコルを説明する。即ち、基本構成要素のお互いのインタラクションによって、ユーザログイン認証とクライアントプロセスとサーバプロセス間の相互認証がどのように行われるかを説明する。以下ではプロトコルを説明するが、その要約は図11.11に示される。説明において、A、G、CおよびSはそれぞれ認証サーバ、チケット承認サーバ、クライアント及びアプリケーションサーバを表す。さらに、K_a、K_g、K_sをそれぞれA、G、Sの秘密鍵とし、K_cをCの秘密鍵とする (K_cはユーザのパスワードから一方向関数を用いて生成される)。

1. ログイン名を入力することによって、ユーザがワークステーションにログオンする時、ログインプログラムはメッセージm_1によって、認証サーバに要求を送り、チケット承認チケットとして知られるものを求める。メッセージm_1はユーザのID(ログイン名)(ID_c)とノンスN_1を含む。このノンスは応答の正当性を確認するのに用いられる。このメッセージは平文形式で送られる。

2. メッセージm_1を受け取ると、認証サーバは認証データベースからこのユーザのパスワードを抽出する。次に、セッション鍵 (K_1) として用いられる乱数を発生する。この後、チケット承認チケットを作成する。これには、ユーザID(ID_c)、チケット承認サーバのID(ID_g)、そのチケットの有効性開始時刻 (T_{s1}) と有効性終了時刻 (T_{e1})(通常は8時間程度) 及びセッション鍵 (K_1) のコピーが含まれる。チケット承認チケットを時間に敏感にすることによって、許可されていないユーザがそれを獲得し後で使うことを防ぐことができる。認証サーバは、チケット承認サーバの秘密鍵 (K_g) を用いてこのチケットを暗号化し、暗号文$C_1 = E((ID_c, ID_g, T_{s1}, T_{e1}, K_1), K_g)$を生成する。この暗号化によって、誰も (当該クライアントも) このチケット承認チケットを改ざんすることができず、ケルベロスサーバのみが復号できることを保証する。次に、クライアントの秘密鍵 (K_c)(ユーザのパスワードから生成される) を用いて、もう1つの暗号文$C_2 = E((N_1, K_1, C_1), K_c)$を生成する。最後に、メッセージ$m_2$で$C_2$をログインプログラムに送り返す。

3. メッセージm_2を受け取ると、ログインプログラムはユーザに対してパスワードを要求する。入力されたパスワードは、一方向関数を通してクライアン

トの秘密鍵 (K_c) に変換される。K_c を獲得した直後に、当該パスワードはコンピュータのメモリから削除され、クライアントのクラッシュによってパスワードが明らかとなってしまう機会を最小限に押さえる。続いて、ログインプログラムは、K_c を用いて C_2 の復号を試みる。ユーザが正しいパスワードを供給していれば、C_2 の復号に成功し、ログインプログラムはメッセージからノンス、セッション鍵及び暗号化されたチケットを獲得する。ノンスによって応答の正当性を確認し、チケット承認サーバとの通信に備えて、セッション鍵と暗号化されたチケットを保管しておく。以上が完了すると、クライアントの秘密鍵もメモリから消去される。このチケットがユーザの認証に役立つからである。このようにして、ユーザワークステーションにおいて、当該ユーザのログインセッションが開始される。

ユーザ認証のために、パスワードがネットワーク上を流れる必要がないことに留意する。さらに、もし侵入者が応答メッセージを横取りしても、それを復号できず、その結果、その中のセッション鍵やチケットを獲得することはできない。

4. 認証されたユーザのためにクライアントワークステーション上で走行しているクライアントプロセスがアプリケーションサーバにアクセスしようとする時、チケット承認サーバに対して、アプリケーションサーバとの通信に用いることができるサービス承認チケットを要求する。このために、クライアントはクライアント ID(ID_c) とタイムスタンプ (T_1) を含む認証関数を作る。この認証関数をセッション鍵 (K_1) を用いて暗号化し、暗号文 $C_3 = E((ID_c, T_1), K_1)$ を得る。再利用可能なチケット承認チケットとは異なり、この認証関数は 1 回限りの使用を目的としており、また非常に短い有効期間しか持たない (通常は数分のオーダー)。次にクライアントは、暗号化された認証関数 (C_3)、暗号化されたチケット承認チケット (C_1)、アプリケーションサーバ ID(ID_s) 及びノンス (N_2) をメッセージ m_3 に入れてチケット承認サーバに送る。

5. メッセージ m_3 を受け取ると、チケット承認サーバはその秘密鍵 K_g を用いて C_1 を復号し、その時点の時刻と T_{e1} を比較して、そのメッセージが期限切れでないことを確認する。また、メッセージからセッション鍵 K_1 を抽出し、それを用いて C_3 を復号して ID_c と T_1 を得る。得られた ID_c は、チケット承認チケットの中の ID_c と比較され、その要求元を認証する。一方、T_1 はその要求が最新のものであることを保証する。すべての確認が成功すれば、チケッ

ト承認サーバはチケットの送信者が、そのチケットの真の所有者であることを確信する。

ここで、クライアントの身元を証明するのは認証関数であり、チケット承認チケットではないことに注意するする必要がある。したがって、チケット承認チケットを用いることは、単に鍵を安全に配布するためである。さらに、認証関数は1回しか用いられず、有効期間も極めて短いので、侵入者が、後の使用のために、チケット承認チケットと認証関数の両方を盗むことはほとんど不可能である。クライアントは、チケット承認サーバに新しいサービス承認チケットを要求するたびに、再利用可能なチケット承認チケットと新しい認証関数を送る。また、チケット承認チケットが再利用可能なことによって、単一ログイン機能のサポートが可能となることにも注意を要する。ユーザは、新しいサービスにアクセスするたびに、パスワードを入力する必要はない。

クライアントの認証が成功した後、チケット承認サーバは新しいランダムなセッション鍵 K_2 を生成し、要求されたサーバへのアクセス用の再利用可能なサービス承認チケットを生成する。このチケットは、クライアント ID(ID_c)、アプリケーションサーバ ID(ID_s)、チケットの有効性開始時刻 (Ts2) と有効性終了時刻 (Te2) 及び新セッション鍵 (K_2) を含む。このサービス承認チケットはアプリケーションサーバの秘密鍵 (K_s) を用いて暗号化され、暗号文 $C_4 = E((ID_c, \mathrm{IDs}, \mathrm{Ts2}, \mathrm{Te2}, K_2), K_s)$ を得る。この暗号化によって、誰も (クライアントさえも) サービス承認チケットを改ざんすることができず、アプリケーションサーバ又はケルベロスサーバのみが復号できることを保証する。次に、チケット承認サーバは古いセッション鍵 (K_1) を用いて、もう1つの暗号文 $C_5 = E((N_2, K_2, C_4), K_1)$ を生成する。メッセージ m_4 によって、この C_5 をクライアントに送り返す。

6. メッセージ m_4 を受け取ると、クライアントは古いセッション鍵 (K_1) を用いて C_5 を復号し、ノンス、新しいセッション鍵及び暗号化されたサービス承認チケットを得る。ノンスを用いて応答の正当性を確認し、アプリケーションサーバとの通信に備えて、新しいセッション鍵と暗号化されたチケットを保存しておく。これで、クライアントはアプリケーションサーバに対して要求メッセージを発行する準備ができたことになる。しかし、相互認証が望ましい場合には、トランザクションやサービス要求に進む前に、クライアントは認証関数を生成する。この認証関数はクライアント ID(ID_c) とタイムスタンプ (T_2) を含んでいる。クライアントはこの認証関数を、セッション鍵 (K_2) を用いて

暗号化し、暗号文 $C_6 = E((ID_c, T_2), K_2)$ を得る。次にクライアントは、メッセージ m_5 に入れて、暗号化された認証関数 (C_6)、暗号化されたサービス承認チケット (C_4) 及び「認証関数から得られるタイムスタンプに 1 を加え、セッション鍵の中に暗号化された応答を返す」というサーバへの要求を送る。

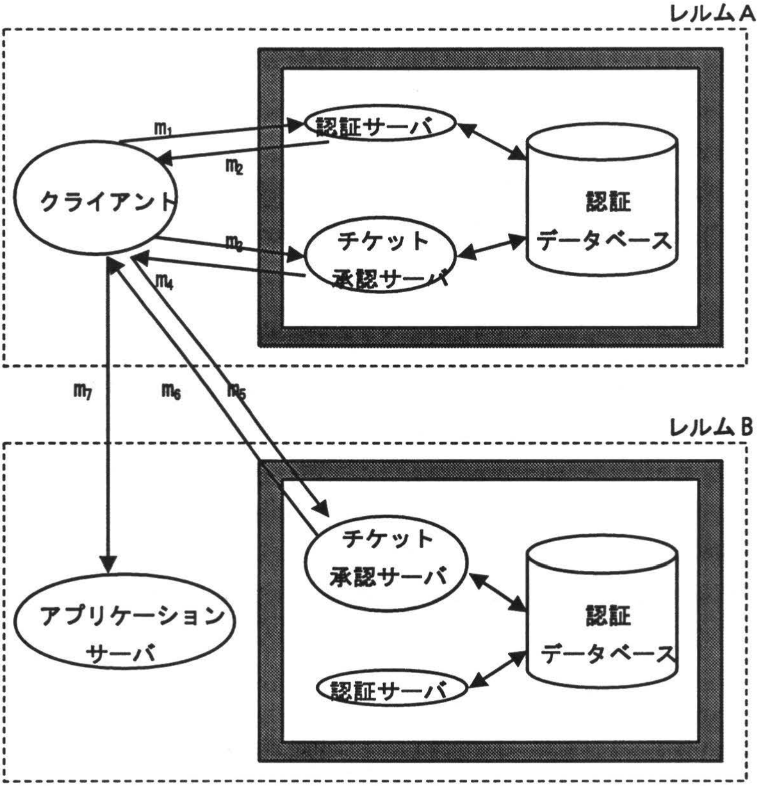
7. メッセージ m_5 を受け取ると、アプリケーションサーバは秘密鍵 K_s を用いて C_4 を復号し、セッション鍵 (K_2) のコピーを抽出する。次に、それを用いて C_6 を復号し、ID_c と T_2 を得る。さらに、T_2 に 1 を加えた値 (T_2) を、セッション鍵 K_2 を用いて暗号化し、暗号文 $C_7 = E(T_2, K_2)$ を得る。これをメッセージ m_6 に入れて、クライアントに送り返す。

8. メッセージ m_6 を受け取ると、クライアントはセッション鍵 K_2 を用いて C_7 を復号し、T_3 を得る。もし $T_3 = T_2 + 1$ であれば、クライアントはサーバが本物であると確信する。この相互認証手続きによって、侵入者がクライアントからアクセス情報を得ようとして、サーバになりすます可能性を防ぐことができる。

この処理過程の結論として、クライアントとサーバは両者の間に安全な通信チャネルが確立されたことを確信する。また、両者は共にセッション鍵 (K_2) を知っており、もし必要なら、これを将来のメッセージ暗号化に使用することができる。

ケルベロスにおけるレルム間認証

組織の境界をまたがるようなシステムでは、1 つのケルベロスサーバにすべてのユーザとサーバを登録するのは適当ではない。したがって、このような環境では、複数のケルベロスサーバが存在し、それぞれが、システム内のユーザ群とサーバ群の部分集合に責任を持つ。ユーザとサーバの部分集合とそれらが登録されたケルベロスサーバを含めて、ケルベロスでは**レルム** (realm) と呼ばれる。典型的な実装では、異なる組織に属するユーザとサーバのネットワークは、通常異なったレルムを構成する。このような環境では、レルム間認証機能が必要であり、それによって、クライアントは異なるレルムに属するサーバと安全に通信することが可能となる。このために、あるレルムのケルベロスサーバは、(秘密鍵を共有して) 他のすべてのレルムのケルベロスサーバに登録される。レルム間認証プロトコルを図 11.12 に示し、以下ではその説明を行う。



m_1 = チケット承認チケットの要求

m_2 = m_1への応答

m_3 = 遠隔チケット承認サーバにアクセスするためのサービス承認チケット要求

m_4 = m_3への応答

m_5 = 遠隔アプリケーションサーバへアクセスするためのサービス承認チケット要求

m_6 = m_5への応答

m_7 = アクセス要求

図 **11.12**　Kerberos のレルム間認証プロトコル

1. 異なるレルムのサーバにアクセスしたいクライアントは、まず他のレルムのチケット承認サーバにアクセスするために、自身のチケット承認サーバからサービス承認チケットを求める要求を行う。

2. 獲得したサービス承認チケットを用いて、クライアントは他のレルムのアプリケーションサーバをアクセスするために、そのレルムのチケット承認サーバからもう 1 つのサービス承認チケットを求める要求を行う。

3. 新しく獲得したサービス承認チケットを用いて、クライアントは、他のレルムのアプリケーションサーバに直接要求を送る。遠隔サーバはそのクライアントの要求を承認するかどうかを選択する。

上述のレルム間認証方法において、各ケルベロスレルムが他のすべてのケルベロスレルムと協調動作することが望ましい場合には、システム内のレルムの数を n とすると、n(n-1)/2 回の鍵交換を行わなければならない。したがって、この方法では規模透過性が十分に達成できない。ネットワークやケルベロスサーバ自体に大きなオーバヘッドを生じる。したがって、ケルベロスの実装では数多くの小さなレルムではなく、少数の比較的大きなレルムを用いるべきである。ケルベロスの最新版 (第 5 版) では、多段ホップレルム間認証もサポートされており、鍵を階層的に共有することが可能となっている [Neuman and Theodore 1994]。

1 つのレルム内では、そのレルムのケルベロスサーバが、システムの滞りない機能のための重要構成要素である。したがって、ケルベロスサーバの信頼性を保証するために、ケルベロスではケルベロスサーバの複製を提供している。複製が行われた時には、すべての複製ケルベロスサーバの認証データベースの一貫性を維持するために、単純なマスタースレーブ方式が用いられる。即ち、すべての変更は、単一のケルベロスデータベース管理サーバ (KDBMS) によってマスターコピーのみに対して行われる。KDBMS はマスターケルベロスサーバマシンの上のみで走行する。ユーザの追加／削除やユーザからのパスワード変更要求などの管理業務は KDBMS によって行われる。マスターケルベロスサーバの認証データベースへの変更は周期的にスレーブケルベロスサーバの認証データベースに伝達される。

ケルベロスの限界

その人気にもかかわらず、ケルベロスはセキュリティに関する完全な解ではない。それは、以下のような理由による [Bellovin and Merritt 1990、Neuman and Theodore 1994]。

1. ケルベロスはパスワード推測攻撃に対しては有効ではない。ユーザが容易に推測できるようなパスワードを選んだ場合には、そのパスワードを推測した侵入者はそのユーザになりすますことができる。侵入者がユーザのパスワードを知るもう1つの方法は、ログインプログラムを修正(トロイの馬)して、ユーザワークステーションに置くことである。この場合にも、侵入者はユーザになりすますに十分な情報を得ることができる。これらの限界に対処するためには、ケルベロスにワンタイムパスワード技法を組み合わせるべきである。ケルベロスにワンタイムパスワードを組み合わせた商用製品が利用可能である。

2. ケルベロスプロトコルは、システム内のノード時刻の緩やかな同期に依存している。この要求を満足することはそれほど困難ではないが、問題は、この目的に使用される同期プロトコル自体が、セキュリティ攻撃に対して安全でなければならないことである。同期時刻に依存しないケルベロスプロトコルは [Kehne 1992] に紹介されている。

3. ケルベロスのもう1つの問題は、チケットの適当な有効期間を決めることが難しいことである。通常は、数時間に制限されている。より良いセキュリティのためには、この有効期間を短くし、登録されていないか又はレベルを下げられたユーザが長時間に渡って資源を利用できないようにすべきである。しかし、ユーザ透過性のためには、有効期間は最長のログインセッションと同程度の長さとすべきである。これは、期限切れのチケットを持ったユーザは、サービス要求を拒否されることになるからである。一旦拒否されると、ユーザはログインセッションを再認証しなければならず、使用するすべてのサービスに新しいサービスチケットを要求しなければならないからである。再認証過程のために任意の時点でアプリケーションに割り込むことは、商用環境では受け入れられない。

4. 最後に、クライアントサーバアプリケーションは、ケルベロス認証を活用するためには、修正しなければならない。この処理は**ケルベロス化** (Kerberization) と呼ばれる。アプリケーションのケルベロス化はケルベロスを

設置する上で、最も困難な部分である。多くの大組織では、アプリケーションのケルベロス化はほとんど不可能と判断し、他の対決策へ向かっている。幸運なことに、時とともにケルベロス化されたアプリケーションの利用可能性が向上しており、今後さらに向上すると期待されている。現在、より多くのメーカが、自分たちの人気製品をケルベロス対応版を製造しつつある。

11.5　アクセス制御

一旦、ユーザやプロセスが認証されると、セキュリティにおける次のステップは、アクセスを許可されていない資源や情報へのアクセスを禁止する方法を工夫することである。この問題は、**認定** (authorization) と呼ばれ、アクセス制御機構を用いて対処される。分散システムで用いられるアクセス制御機構 (保護機構としても知られる) は、基本的に集中型システムのそれと同じである。主な相違点は、集中型システムではすべての資源が注意深く配置されるので、アクセス制御は中央集権的に行うことができる。しかし、分散クライアントサーバ環境では、各サーバが自分自身の資源へのアクセス制御に責任を持つ。

コンピュータシステムのアクセス制御を語る時、以下の用語を使用するのが普通である。

1. **オブジェクト** (object):オブジェクトは、それに対するアクセスが制御されるような実体である。オブジェクトは、プロセス、ファイル、データベース、セマフォ、木構造データなどの抽象的な実体でも良いし、CPU、メモリセグメント、プリンタ、カードリーダ、テープドライブ、ネットワークサイトなどの物理的な実体でも良い。

各オブジェクトは、システム内の他のオブジェクトと区別ができるように唯一の名前を持つ。さらに、オブジェクトには、それに対して実行可能な操作を決める「型」が付随している。例えば、「データファイル」という型に属するオブジェクトに対して実行可能な操作の集合は、Open、Close、Delete、Read および Write であり、「プログラムファイル」という型に属するオブジェクトに対しては、Read、Write および Execute である。同様に、「セマフォ」という型のオブジェクトに対して可能な操作は、Up と Down であり、「テープドライブ」という型のオブジェクトに対しては、Read、Write、Rewind である。

2. **サブジェクト** (subject):サブジェクトは活動的な実体であり、オブジェク

トに対するそのアクセスが制御されなければならない。即ち、オブジェクトへの操作を実行する意志を持ち、アクセス許可が得られた実体をサブジェクトと呼ぶ。サブジェクトの例はプロセスやユーザである。保護されなければならないという意味で、サブジェクトもオブジェクトであることに留意する。したがって、各サブジェクトは唯一の名前を持つ。

3. **保護規則** (protection rule):保護規則とは、サブジェクトとオブジェクトがインタラクションを行う方法を定義するものである。即ち、保護規則がオブジェクトに対するサブジェクトのアクセスを支配する。したがって、各 (サブジェクト、オブジェクト) 組には**アクセス権** (access right) が付随し、これが、サブジェクトがそのオブジェクトに対して行い得る操作 (そのオブジェクトの型に関する操作) の部分集合を定義する。システムのアクセス権の完全集合は、どのサブジェクトがどのオブジェクトに対してどの操作を行うことができるかを定義する。この集合は、任意の時点で、その時点のシステムの**保護状態** (protection state) を定義する。

サブジェクトのオブジェクトへのアクセスを制御するための、保護規則の厳密な適用方法はシステムが用いるアクセス制御モデルに依存する。以下のアクセス制御モデルが提案されている。

1. アクセス行列モデル [Lampson 1971、Graham and Denning 1972、Harrison et al. 1976]
2. 情報フロー制御モデル [Denning 1976、Bell and LaPadula 1973]
3. セキュリティカーネルモデル [Ames et al. 1983、Rushby and Randell 1983]

これらの中で、アクセス行列モデルが最もよく知られたものであり、既存の集中型システムや分散システムで広く用いられている。他の 2 つのモデルは、主に理論的な興味を引き起こすだけである。以下では、アクセス行列モデルに基づくアクセス制御機構について説明する。

11.5.1 保護ドメイン

最小特権の原則は、任意の時点においてサブジェクトがその仕事を完成するためにその時点で必要なオブジェクトのみにアクセスできることを要求してい

る。したがって、時間の経過に伴って、サブジェクトはオブジェクトへのアクセス権の集合を変更する必要がある。この集合は、任意の時点で、行わなければならない特定の仕事に依存する。ドメインという概念は、セキュリティシステムにおいて、アクセス制御に関するこの種の柔軟性を提供するためによく用いられるものである。

ドメイン (domain) はアクセス権の集合の抽象的な定義であり、(オブジェクト、権利) 組として定義される。各組はオブジェクトとそのオブジェクトに対する 1 つ以上の操作を規定する。許可されている各操作は**権利** (right) と呼ばれる。

ドメインの概念に基づくセキュリティシステムは、ドメインの集合を定義し、各ドメインはそれぞれの (オブジェクト、権利) 組を持つ。任意の時点で、プロセスのアクセス権は、その時点で、そのプロセスが存在するドメインで定義されたアクセス権と同じである。プロセスは、実行中に、あるドメインから他のドメインに移動することもできる。

図 11.13 は 3 つの保護ドメイン D_1、D_2、D_3 を持つシステムの例を示している。この例から、以下のような点が読み取れる。

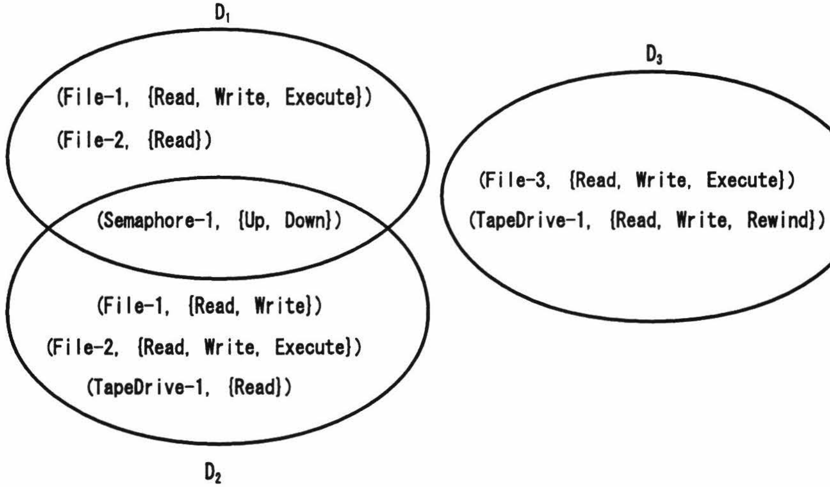


図 11.13　3 つの保護ドメインを持つシステム

1. ドメインは必ずしも分離されている必要はない。即ち、同じアクセス権が同

時に2つ以上のドメインに存在してもよい。例えば、アクセス権 (Semaphore-1, Up, Down) は、D_1 と D_2 の両方に存在する。このことは、いずれかのドメインに存在するサブジェクトが Semaphore-1 に対して、Up および Down 操作を行えることを意味している。

2. 同じオブジェクトが、異なるドメインに属し、それぞれのドメインで異なった権利を有することができる。例えば、File-1 と File-2 に関する権利は、D_1 と D_2 で異なり、TapeDrive-1 に関する権利は、D_2 と D_3 で異なる。File-1 に関しては、ドメイン D_1 に存在するサブジェクトは、Read、Write および Execute 操作を行うことができるが、ドメイン D_2 に存在するサブジェクトは Read 操作と Write 操作のみを行える。したがって、File-1 を実行するためには、サブジェクトはドメイン D_1 に存在しなければならない。

3. オブジェクトが1つのドメインに存在する場合には、そのドメイン内のサブジェクトによってのみアクセス可能である。例えば、File-3 はドメイン D_3 内のサブジェクトによってのみアクセスされる。

ドメインは抽象的な概念であるため、その実現とドメイン変更の規則はシステム依存である。例えば、以下のような方法でドメインを実現することができる。

1. 各ユーザをドメインとする。この場合、プロセスには、実行目的のユーザの身元に従って、ドメインが割り当てられる。ドメイン変更は、あるユーザがログアウトし、他のユーザがログインした時に発生する。

2. 各プロセスをドメインとする。この場合、プロセスのアクセス権は、それ自身のドメインのアクセス権に限定される。ドメイン変更は、あるプロセスがメッセージを他のプロセスに送り、応答を待つ時に発生する。

3. 各手続きをドメインとする。この場合、各手続きは一式のアクセス権を持ち、ドメイン変更は、ある手続きが実行中に他の手続きを呼び出した時に発生する。

UNIX の保護方式では、ユーザ指向ドメインと手続き指向ドメインの両

方の概念が使用されている。まず、ユーザ指向ドメインの概念の使用から説明する。UNIX では、各ユーザに**ユーザ ID** (user-id)(uid) と**グループ ID** (group-id)(gid) が付随する。(uid, gid) 組は、オブジェクトのリストと、それらに対して行える操作の型を特定する。プロセスのドメインは、そのプロセスを使用するのユーザに依存する。したがって、プロセスのドメインは、その (uid, gid) 組で定義される。即ち、同じ (uid, gid) 組を持つ 2 つのプロセスは、全く同じアクセス権を持ち、異なる (uid, gid) 組を持つ 2 つのプロセスは、異なるアクセス権を持つ。プロセスは、setuid コマンド又は setgid コマンドを実行して、新しい uid 又は gid を獲得することによって、ドメインを遷移する。さらに、UNIX における**スーパーユーザ** (superuser) は、人のことではなく、システムのすべてのオブジェクトへのアクセスを必要とする、最も特権的なプロセスが走行する特別のドメインの名前であることを付言しておく。

次に、UNIX における手続き指向ドメインの概念の使用について説明する。UNIX では、すべての手続きは 2 つのクラスに分類される。ユーザ手続きとカーネル手続きである。これらの 2 つのクラスが、**ユーザモード** (user mode) と**カーネルモード** (kernel mode) と呼ばれる 2 つのドメインを構成する。カーネルモードで走行するプロセスは、ユーザモードで走行するプロセスと比べて、異なったアクセス権を有する。例えば、カーネルモードでは、プロセスは物理メモリのすべてのページ、ディスクの全領域及び他のすべての保護資源にアクセス可能である。この 2 ドメインアーキテクチャでは、ユーザプロセスがシステム呼び出しを行った時に、ユーザモードからカーネルモードに遷移する。したがって、特定の時点で、あるプロセスのドメインとアクセス権は、その時点でユーザ手続きを実行しているのか、又はカーネル手続きを実行しているのかに依存する。

Multics[Schroeder et al. 1977] は、より一般的な形の手続き指向ドメインを用いていた。2 つのドメイン (ユーザモードとカーネルモード) を用いる UNIX とは異なって、Multics アーキテクチャは最大 64 までのドメインをサポートするという柔軟性を有していた。Multics の各ドメインは、**リング** (ring) と呼ばれていた。図 11.14 に示すように、リングは同心円状になっており、最も内側のリングにある手続きはオペレーティングシステムカーネルである。これが最も強力であり、最大限のアクセス権を有する。最も内側のリングから外側へ向かうに従って、リングは次第に弱くなり、より少数のアクセス権しか持たなくなる。実際、リング i のアクセス特権はリング j のアクセス特権の部分集合である。ここで、$i > j$ であり、リングの番号は最も内側のものから出発して、外

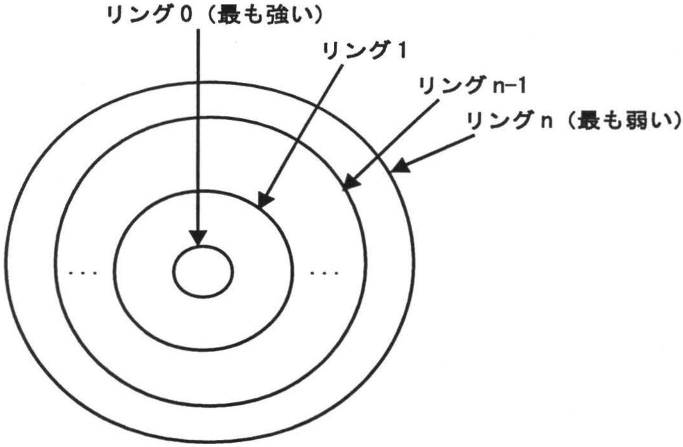


図 11.14 Multics 保護ドメインのリングアーキテクチャ

へ向かうに従って増大するものとする。即ち、$j < i$ は、リング j の手続きがリング i の手続きよりも多くのアクセス権を有することを意味する。プロセスは、その生存期間中に、複数のドメイン (リング) の中で動作することができる。ドメイン遷移は、あるドメイン (リング) の手続きが、他のドメイン (リング) の手続きを呼び出した時に発生する。明らかに、ドメイン遷移は制御された状態で行われる。さもなければ、任意のプロセスが最も内側のリングで実行を開始することができ、それに対する保護ができないことになってしまうからである。

11.5.2 アクセス行列

ドメインに基づく保護方式では、特定のドメインに、どのオブジェクトに対するどの権利が属しているかを追跡しなければならない。アクセス行列モデルでは、この情報は行列として表現され、それは**アクセス行列** (access matrix) と呼ばれる。アクセス行列は以下のような形をとる。

1. 行はドメインを表す。
2. 列はオブジェクトを表す。
3. アクセス行列の要素は、一式のアクセス権から成る。
4. アクセス行列の第 (i, j) 要素は、ドメイン D_i で実行されているプロセスがオブジェクト O_j に対して行える操作の集合を定義する。

オブジェクト / ドメイン	F_1 ファイル 1	F_2 ファイル 2	F_3 ファイル 3	S_1 セマフォ 1	T_1 テープドライブ 1
D_1	Read Write Execute	Read		Up Down	
D_2	Read Write	Read Write Execute		Up Down	Read
D_3			Read Write Execute		Read Write Rewind

図 11.15　図 11.13 のシステムの保護状態に関するアクセス行列

任意の時点で、システムの**保護状態** (protection state) がアクセス行列の内容によって定義される。図 11.15 のアクセス行列は、図 11.13 のシステムの保護状態を示している。

システムの保護状態を表すのにアクセス行列が用いられる場合、以下のような課題が解決されなければならない。

1. アクセス行列要素の内容をどのように決定するか。
2. サブジェクトによるオブジェクトへのアクセスをどのように正当化するのか。
3. 制御された形で、どのようにサブジェクトのドメイン遷移を可能とするか。
4. 制御された形で、どのようにシステムの保護状態の変更を可能とするか。

これらの課題に対する通常の対処方法を以下で説明する。

アクセス行列要素の内容の決定

(i, j) 要素にどの権利を含めるべきかという方針決定はシステム依存である。しかし一般的には、アクセス行列のエントリ内容のうち、ユーザ定義オブジェクトに対応するものはユーザが決定し、システム定義オブジェクトに対応する

ものはシステムが決定する。例えば、ユーザが新しいオブジェクト O_j を生成した時には、そのオブジェクトに関するアクセス制御のユーザ仕様によって決定されるような適当なエントリを持つ列 j がアクセス行列に追加される。

サブジェクトによるオブジェクトへのアクセスの正当化

アクセス行列が与えられた時、どのようにして、行列の保護状態で許されたような方法のみで、サブジェクトによるオブジェクトへのアクセスが可能となるのか。これに関しては、オブジェクトの各型に**オブジェクトモニタ** (object monitor) を付随し、サブジェクトによるオブジェクトへのすべてのアクセスが以下のように正当化される。

1. ドメイン D 内のサブジェクト S が、オブジェクト O に対する r アクセスを開始する。ここで、r は、O に対して行われる可能性のある操作の集合に属する。

2. 保護システムはベクトル (D, r, O) を形成し、それを O のオブジェクトモニタに渡す。

3. O のオブジェクトモニタは、アクセス行列の (D, O) 要素の中で操作 r を探す。もし r があれば、アクセスは許可されるが、さもなければ保護違反が発生する。

制御された形でのドメイン遷移

最小特権の原則によって、プロセスはその生存期間中に、あるドメインから他のドメインに遷移することができるべきであり、それによって、任意の時点において、その時点で仕事を行うのに必要なだけの権利を与えられるべきである。しかし、プロセスのドメイン遷移は制御された形で行われるなければならない。さもなければ、あるプロセスが強力なドメインに遷移し、システムの保護方針に違反するかもしれない。

ドメイン遷移は、可能な操作が switch のみであるようなオブジェクトとしてドメインを扱うことによって制御される。したがって、ドメイン遷移が制御された形で行えるようにするために、ドメインもアクセス行列のオブジェクトに含まれる。

図 11.16 は、図 11.15 のアクセス行列に、3 つのドメインをオブジェクトとして付加したものである。この修正アクセス行列において、ドメイン D_i からドメイン D_j への遷移は、アクセス行列の (D_i, D_j) 要素に権利 switch がある場

合に限って許可される。図 11.16 では、ドメイン D_1 で実行されているプロセスは、ドメイン D_2 又はドメイン D_3 への遷移が可能であり、ドメイン D_2 で実行されているプロセスは、ドメイン D_3 への遷移のみが可能である。また、ドメイン D_3 で実行されているプロセスは、他のどのドメインにも遷移できない。

ドメイン ＼ オブジェクト	F_1	F_2	F_3	S_1	T_1	D_1	D_2	D_3
D_1	Read Write Execute	Read		Up Down			Switch	Switch
D_2	Read Write	Read Write Execute		Up Down	Read			Switch
D_3			Read Write Execute		Read Write Rewind			

図 11.16　ドメインをオブジェクトとして持った図 11.15 のアクセス行列

制御された形での保護状態の変更

柔軟な設計では、保護システムはドメインの内容の変更も可能とするべきである。しかし、この機能は必須のものではない。もし、ドメインの内容が変更できないなら、変更された内容を持つ新しいドメインを生成し、ドメイン内容の変更を行いたい時には、この新しいドメインに遷移することによって、同じ効果が得られるからである。必須ではないが、この保護状態の制御された形での変更を可能とする機能は、より優れた柔軟性のために、保護システムで通常提供されるものである。

アクセス行列モデルでは、この機能はアクセス行列自体を保護すべきオブジェクトとして扱うことによって提供される。実際のところ、アクセス行例の各要素は個別に修正可能なため、各要素を保護されるべきオブジェクトとみなさなければならない。制御された形での変更を可能とするために、この新しいオブジェクトに対して可能な権利は、copy、owner および control である。記

オブジェクト ＼ ドメイン	F_1 ファイル1	F_2 ファイル2	F_3 ファイル3
D_1	Read* Write* Execute	Read	
D_2	Read* Write	Read* Write Execute*	
D_3			Read Write Execute

図 **11.17** コピー権を持ったアクセス行列

述を簡単にするために、アクセス行列の要素の内容に対して可能な変更を 2 つのタイプに分類する。列要素に対して可能な変更と、行要素に対して可能な変更である。

列要素に対して可能な変更

copy 権と owner 権によって、プロセスはある列の要素を変更することができる。アクセス権をあるドメインから他のドメインへコピーできることは、アクセス権に星印 (*) を付加することによって表す。例えば、図 11.17 は図 11.15 における最初の 3 つのオブジェクト (F_1、F_2、F_3) のアクセス権を、幾つかのコピー権とともに示したものである。ドメイン D_1 で実行されているプロセスは、F_1 に対する Read 操作と Write 操作を、他の任意のドメイン (列 F_1 の任意のエントリ) にコピーすることができる。一方、ドメイン D_2 で実行されているプロセスは、F_1 に対する Read 操作と、F_2 に対する Read 操作及び Execute 操作を、他の任意のドメインにコピーすることができる。F_3 に対する操作は、どれも他のドメインにコピーすることはできない。

コピー権には以下のような 3 つの変形がある。

1. 転送 (transfer):この場合、権利がアクセス行列の第 (i, j) 要素から第 (k, j)

要素にコピーされた時、第 (i, j) 要素からは削除される。

2. **伝搬禁止コピー** (copy with propagation not allowed):この場合、権利 $R*$ が第 (i, j) 要素から第 (k, j) 要素にコピーされた時、第 (k, j) 要素には権利 R のみが生成される (権利 $R*$ ではない)。このことは、ドメイン D_k で実行されているプロセスが権利 R をさらにコピーすることはできないことを意味している。

3. **伝搬可能コピー** (copy with propagation allowed):この場合、権利 $R*$ が第 (i, j) 要素から第 (k, j) 要素にコピーされた時、第 (k, j) 要素には権利 $R*$ が生成される。その結果、ドメイン D_k で実行されているプロセスは権利 $R*$ 又は R をさらにコピーすることができる。

一方、owner 権は列要素の権利の追加／削除を制御された形で可能とするために用いられる。アクセス行列の第 (i, j) 要素に owner 権が含まれている場合、ドメイン D_i で実行されているプロセスは、第 j 列の任意の要素内の任意の権利を追加／削除することができる。図 11.18 は、図 11.17 のアクセス行列の 3 つのオブジェクトに owner 権を持たせたものである。この図では、ドメイン D_1、D_2、D_3 は、それぞれオブジェクト F_1、F_2、F_3 に owner 権を持つ。したがって、ドメイン D_1 で実行されているプロセスは、列 F_1 の任意の要素内にある F_1 に関する正当な権利を追加／削除することができる。同様に、ドメイン D_2 で実行されているプロセスは、列 F_2 の任意の要素内にある F_2 に関する正当な権利を追加／削除することができ、ドメイン D_3 で実行されているプロセスは、列 F_3 の任意の要素内にある F_3 に関する正当な権利を追加／削除することができる。

行要素に対して可能な変更

control 権はドメインオブジェクトに対してのみ適用可能であり、プロセスが行の要素を変更することを可能とする。アクセス行列の第 (D_i, D_j) 要素に control 権が存在する場合、ドメイン D_i で実行されているプロセスは、行 D_j から任意のアクセス権を削除することができる。例えば、図 11.19 は図 11.16 のアクセス行列に、control 権を含めたものである。この図では、(D_1, D_2) 要素と (D_1, D_3) 要素が control 権を持っているため、ドメイン D_1 で実行されているプロセスは、行 D_2 と D_3 の任意の権利を削除することができる。同様に、(D_2, D_3) 要素が control 権を持つことから、ドメイン D_2 で実行されているプロセスは、行 D_3 の任意の権利を削除することができる。

オブジェクト / ドメイン	F_1 ファイル1	F_2 ファイル2	F_3 ファイル3
D_1	Read* Write* Execute Owner	Read	
D_2	Read* Write	Read* Write Execute* Owner	
D_3			Read Write Execute Owner

図 **11.18** 所有権を持ったアクセス行列

11.5.3 アクセス行列の実装

実際には、アクセス行列は巨大かつ疎である。大部分のドメインは、ほとんどのオブジェクトに対して全くアクセス権を持たない。即ち、大部分の要素が空である。したがって、アクセス行列を直接2次元行列として実装することは非常に効率が悪く、高価である(ディスク領域の浪費)。さらに、分散システムでは、サブジェクトとオブジェクトが異なったサイトに置かれる可能性があり、このことが実装課題をより一層複雑なものにする。現在の分散システムにおけるアクセス行列の実装に関して、最もよく用いられている方法は、アクセス制御リスト(ACL)とケーパビリティである。例えば、Andrew[Satyanarayanan 1989]、Apollo[Levine 1986]及びButler[Dannenberg and Hibbard 1985]はACLを用いており、Accent[Rashid and Robertson 1981]、Amoeba[Mullender and Tanenbaum 1986]及びMach[Sansom et al. 1986]はケーパビリティを用いている。以下では、これらの2つの方法について説明する。

オブジェクト / ドメイン	F_1	F_2	F_3	S_1	T_1	D_1	D_2	D_3
D_1	Read Write Execute	Read		Up Down			Switch Control	Switch Control
D_2	Read Write	Read Write Execute		Up Down	Read			Switch Control
D_3			Read Write Execute		Read Write Rewind			

図 11.19　制御権を持ったアクセス行列

アクセス制御リスト

この方法では、アクセス行列は列単位に分割される。行列の各列は、その列に対応するオブジェクトに関するアクセスリストとして実装される。行列の空の要素は、アクセスリストには格納されない。したがって、各オブジェクトに関して、順序対 (ドメイン、権利) のリストが管理され、これが、対象となっているオブジェクトに関する空でないアクセス権集合をもつドメインを定義する。ACL に基づくセキュリティシステムのさらに詳しい動作と特性に関しては、以下で述べる。

アクセスの正当化

ドメイン D 内のサブジェクトがオブジェクト O に対して操作 r を行う時は常に、まずオブジェクト O のアクセスリストから、ドメイン欄が D であるようなリスト要素を探索する。次に、この要素の権利欄から、操作 r を探索する。もし見つかれば、操作を続行することができるが、さもなければ、保護違反が発生する。

この方法では、各アクセスごとに、アクセスリストが確認される。これは、セキュリティの観点からは極めて望ましい特徴である。しかし、アクセスごとにアクセスリストを参照することは、大きなオーバヘッドを生じる。アクセスリストが長い場合には、特にそうである。この欠点は、活性化されドメインだ

けのアクセスリスト要素のキャッシュを維持管理することで克服できる。

権利の承認

オブジェクト O に関するアクセス権は、以下の手順よってドメイン D に対して承認される。

1. まず、オブジェクト O のアクセスリストから、そのドメイン欄が D であるようなリスト要素が探索される。

2. もし見つかれば、このリスト要素の権利欄に権利 r が追加される。さもなければ、新しいリスト要素が、オブジェクト O のアクセスリストに追加される。このリスト要素のドメイン欄と権利欄は、それぞれ D と r に設定される。

権利の譲渡

オブジェクト O のアクセス権 r は、以下の手順によって、あるドメイン D_1 から他のドメイン D_2 へ譲渡 (伝搬) される。

1. まず、オブジェクト O のアクセスリストを確認して、D_1 がオブジェクト O の owner 権かアクセス権 r の copy 権を有することを保証する。

2. D_1 が上述の 2 つの権利のいずれかを有すれば、上述の手続きによって、オブジェクト O へのアクセス権 r がドメイン D_2 に対して承認される。さもなければ、保護違反が発生する。

権利の取り消し

オブジェクト O に対するアクセス権 r は、O のアクセスリストにおける D の権利集合から単に r を削除することによって、ドメイン D から取り消される。

ファイルの保護に関して、幾つかのシステムでは、(uid, gid) 組がドメインを形成する UNIX のように、ユーザ指向ドメインを用いている。このようなシステムでは、ユーザのアクセス権の削除はさらに複雑になる。アクセス権の取り消しは、ユーザの uid(もし存在すれば) を問題となっているオブジェクトのアクセスリストから削除することを必要とし、また、そのオブジェクトにアクセスするドメインに属するすべてのグループからユーザのメンバーシップをキャンセルすることを必要とするからである。大規模な分散システムでは、

ユーザが削除されるべきすべてのグループを発見する処理や、実際の削除を行う処理は、非常に長時間を要するため、緊急時には受け入れ難いものとなる。この問題を克服するために、**否定権利** (negative right) の概念が用いられる [Satyanarayanan 1989,1990]。この概念は、オブジェクトに対するユーザのアクセス権を取り消すためには、そのユーザにそのオブジェクトに対する否定権利を与えればよいという考え方に基づいている。否定権利とは、指定された権利の否定を示している。衝突がある場合には、否定が優先される。この拡張によって、ACL に否定権利を含めることが可能となり、ユーザ i とユーザ j を除くすべてのユーザがオブジェクト O へのアクセスを実行するというような表現が可能となる。ユーザの肯定権利から、そのユーザの否定権利の和集合を引き去ったものが、実際にそのユーザに与えられているすべての権利である。このように、否定権利は高速で選択的な取り消し機構として働き、特に大規模な分散システムで有用なものである。

ACL の主な利点は、与えられたオブジェクトに関して、それにアクセス可能なドメインを効率良く決定できることである。一方、主な欠点は、与えられたドメインに関して、アクセス権集合を効率良く決定することができないことである。

ケーパビリティ

この方法では、アクセス行列を列単位で分割する代わりに、行単位で分割し、各行はそのドメインに付随する。明らかに、空の要素は無視される。したがって、各ドメインに関して、順序対 (オブジェクト、権利) のリストが維持管理され、これはそのドメインが何らかのアクセス権を有するすべてのオブジェクトを定義する。各 (オブジェクト、権利) 組は**ケーパビリティ** (capability) と呼ばれ、ドメインに付随するリストは**ケーパビリティリスト** (capability list) と呼ばれる。

ケーパビリティは以下の 2 つの目的のために用いられる。

- オブジェクトを一意に特定する
- その所有者が、1 つ以上の許可モードにあることを確認したオブジェクトへのアクセスを可能とする

したがって、図 11.20 に示すように、ケーパビリティは 2 つの基本部分から構成される。即ち、オブジェクト識別子部と権利情報部である。通常、オブ

ジェクト識別子部はオブジェクトへのポインタを含み、そのオブジェクトに関する大域的でユニークなシステム指向の名前として機能する。一方、権利情報部は、オブジェクトに対してこのケーパビリティで実行可能な操作を決めるビットの集合である。ケーパビリティに基づくセキュリティシステムのさらに詳しい動作と特性に関しては、以下で述べる。説明を簡単にするために、以下の記述では、各ユーザ/プロセスがシステムのドメインを形成すると仮定する。

オブジェクト識別子	**権利情報**

図 11.20　ケーパビリティの 2 つの基本部分

アクセスの正当化

ケーパビリティは偽造できないチケットと考えることができ、このチケットの所有者は、1 つ以上の許可モード (権利情報部で指定される) でオブジェクト (オブジェクト識別子部で特定される) にアクセスすることができる。あるケーパビリティを持つプロセスは、そのケーパビリティによって特定されるオブジェクトに対して、そのケーパビリティによって許可されるモードでアクセスすることができる。通常は、同じオブジェクトに複数のケーパビリティが存在する。各ケーパビリティは、所有者に異なるアクセス権を与える。異なった所有者が同じケーパビリティを所有する場合には、それら所有者すべてに同じアクセス権が提供される。

単にケーパビリティを所有するだけで、そのケーパビリティに付随するモードでのアクセスが許可されていることを意味するため、オブジェクト O に対して操作 r を実行するためには、プロセスは、パラメータとしてオブジェクト O のケーパビリティを指定して、操作 r を実行すればよい。オブジェクト O のモニタが (ケーパビリティ付きの) アクセス要求を受け取ったときには、そのケーパビリティの権利情報部が操作 r の許可を持つかどうかを確認するだけでよい。

ケーパビリティに基づくセキュリティシステムでは、アクセスが許されるか否かを確認するのに、リストを探索する必要がないことに留意しておく。セキュリティシステムは、プロセスによって与えられたケーパビリティがオブジェクトに対して要求されている操作に関して正当であるかどうかを確認す

るだけでよい。したがって、プロセスがあるオブジェクトのケーパビリティを一旦所有すれば、セキュリティシステムによるさらなる確認を必要とせず、そのケーパビリティで許されているモードの 1 つで、そのオブジェクトにアクセスすることができる。このような理由から、ケーパビリティに基づくセキュリティシステムは、ACL に基づくセキュリティシステムに比べて、効率的である。ケーパビリティに基づく方法では、ユーザの身元の確認を行えないことに注意する必要がある。アクセスの正当化において、ユーザの身元確認が必要な場合には、何らかのユーザ認証機構を用いなければならない。

権利の承認と譲渡

ケーパビリティに基づくセキュリティ方式では、ユーザがアクセス可能なすべてのオブジェクトとそれに付随するアクセス許可を特定するケーパビリティのリストを、各ユーザが維持管理する。しかし、まず最初に、ユーザはどのようにしてケーパビリティを獲得するのか。

ケーパビリティに基づくシステムでは、各オブジェクト型に対して 1 つ以上のオブジェクト管理者が存在するのが普通である。オブジェクトの生成などの操作要求は、そのオブジェクト型のオブジェクト管理者の 1 つに送られる。新しいオブジェクトが生成された時には、そのオブジェクトを生成したオブジェクト管理者が、そのオブジェクトに対するすべての権利を持つケーパビリティを (オブジェクト生成処理の一環として) 生成する。生成されたケーパビリティは、以降の使用のために、オブジェクトの所有者に返される。次に、その所有者は、そのオブジェクトを共有している他のユーザにそのケーパビリティを与えるかもしれない。しかし、その所有者は、そのオブジェクトを共有するユーザごとに、異なった形でアクセスモードを制限したいかもしれない。したがって、所有者がケーパビリティを他のユーザに渡す前に、幾つかの権利を削除することによって、ケーパビリティを制限する必要があるかもしれない。これを行う通常の方法は、オブジェクト管理者にケーパビリティを制限するような機能を持たせることである。この機能は、呼び出された時に、必要なアクセス許可だけを持つような新しいケーパビリティを生成し、呼び出し側には、その新しく生成されたケーパビリティを返す。次に、このケーパビリティが対象ユーザに渡される。

許可されていないアクセスからのケーパビリティの保護

ケーパビリティに基づくセキュリティシステムにおいて、オブジェクトが許

可されていないアクセスから保護されていることを保証するためには、以下の基本要求を満足しなければならない。

1. ケーパビリティはシステム全体において、オブジェクトを一意に特定できなければならない。与えられたケーパビリティを持つオブジェクトが削除されても、そのケーパビリティが再利用されることがないことが重要である。これは、古いケーパビリティを保持しているユーザがいるかもしれないからである。古いケーパビリティを使用した場合には、異なったオブジェクトにアクセスするのではなく、誤りを発生させるべきである。

2. ケーパビリティはユーザによる変更から保護されなければならない。そのためには、ケーパビリティを、オペレーティングシステムによって維持管理され、ユーザからは間接的にしかアクセスできないような、変更不可能な保護されたオブジェクトとして扱うことが必要である。

3. 正しいケーパビリティを推測することが、不可能又は極めて難しくなければならない。これは、ケーパビリティに基づく方式では、保護の度合いは確率的であり、正しいケーパビリティを推測する困難さに比例するからである。

通常、これらの要求を満足するために、以下のような方法が用いられる。

1. **タグアーキテクチャ**(tagged architecture):この方法では、各オブジェクトは、それがケーパビリティか、又は整数、ポインタ、文字、命令などの通常にアクセス可能なデータかを示すタグを持つ。通常、タグはハードウェアによって実現され、メモリの単位 (通常はワード) にタグが付加される。この場合、各メモリワードはタグ欄を持ち、そのワードがケーパビリティを含むかどうかを示す。タグ欄はアプリケーションプログラムから直接アクセスすることはできず、カーネルモードで走行しているプログラム (即ち、オペレーティングシステム) のみが修正可能である。ケーパビリティかその他のオブジェクトかを区別するには、タグ欄として 1 ビットあればよいが、通常、タグアーキテクチャのマシンでは、ソフトウェアが 2^n 個の異なった型のオブジェクト (メモリ内容) を識別可能なように、n ビットのタグ欄を用いるのが普通である。

2. **分割実装** (partitioned implementation):ケーパビリティの完全性を保持す

るもう1つの方法は、ケーパビリティをデータとは区別して、オペレーティングシステムのみがアクセスできる特別のセグメントに格納することである。これを実装する1つの方法は、ユーザプロセスのアドレス空間を2つの部分に分割し、一方はプロセスからアクセスでき、他方はオペレーティングシステムのみがアクセスできるようにすることである。前者はプロセスの通常のデータと命令を含み、後者はそのケーパビリティを含む。

3. **疎なケーパビリティの暗号化** (encryption of sparse capabilities):許可されていないアクセスからケーパビリティを保護するタグ方式や分割方式は、集中型コンピュータシステム向きのものである。これらの方法は、分散システムでの使用には不向きである。分散システムでは、そこで用いられるセキュリティ機構によって、ケーパビリティを安全にあるノードから他のノードに転送可能でなければならないからである。第3の方法は、ケーパビリティを他のオブジェクトから区別するのにタグや分割を用いないものであり、特に分散システムに適したものである。ケーパビリティをユーザによる変更から守ることに加えて、この方法では、ケーパビリティの唯一性と推測困難性が実現される。

この方法では、ケーパビリティのために十分に疎な巨大な名前空間が用いられる。唯一性は、10.4節で述べた唯一のシステム指向識別子を生成する方法によって達成される。この識別子がケーパビリティのオブジェクト識別子部を形成する。一方、ケーパビリティを推測困難や偽造困難にするために、各ケーパビリティの権利情報部には乱数を含む付加的な欄が組み込まれる。この乱数によって、任意の正当なケーパビリティを生成しようとするユーザの悪意の仕事を、実行不可能なくらいに長大にする。さらに、ケーパビリティがユーザに対して発行される前に、権利情報部と乱数部はオブジェクト管理者によって暗号化される。その秘密鍵はオブジェクト管理者のみが利用可能である。あるプロセスが、オブジェクトアクセスの要求とともにケーパビリティを提示した時、オブジェクト管理者はその鍵を用いてケーパビリティの暗号化された部分を復号し、それから使用する。この方法では、制限された許可を与えるケーパビリティの権利情報部は、その所有者によってより多くの許可を持つケーパビリティへ変更することができない。

権利増幅

権利増幅 (rights amplification) の概念は、Hydra[Cohen and Jefferson 1975]で導入された。ケーパビリティに基づくシステムでは、オブジェクトは

型付けされ、既定の操作の集合を認識することを見てきた。あるオブジェクト型に関する既定の操作の集合は、Hydraでは**補助権利** (auxiliary right) として知られる。補助権利に加えて、各オブジェクト型は一式の**カーネル権利** (kernel right) を持つ。カーネル権利には、オブジェクトのデータ部を扱うget、put、addなどと、オブジェクトのケーパビリティリスト部を扱うload、store、append、delete、copy、createがある。カーネル権利はカーネル内で実装され、ユーザプロセスからは見えない。

あるオブジェクトに対する操作実行要求は、そのオブジェクト型のオブジェクト管理者に送られる。その要求には、パラメータとして、そのオブジェクトのケーパビリティが含まれる。このケーパビリティは、そのオブジェクトに対する何らかの操作を起動する補助権利を含むかもしれないが、そのオブジェクトに関するカーネル権利を含むことはない。オブジェクト管理者自体が通常のプログラムであるため、ここで問題が生じる。要求された操作をうまく実行するためには、オブジェクト管理者がカーネル操作を起動できることは必須である。Hydraの権利増幅技法は、ケーパビリティ自身よりも多くの権利を与えるような権利テンプレートを、オブジェクト管理者に与えることによって、この問題を解決する。オブジェクト管理者に与えられる付加的な権利によって、オブジェクト管理者はそのオブジェクトに対するカーネル操作を実行することができる。

プロセス P がオブジェクト O に対して操作rを起動する場合、オブジェクト管理者 M が操作 r をそのオブジェクトに対して実行する段階で、P より与えられるケーパビリティ C は C_a に増幅される。これは、操作 r を実行するために、M が O を表す記憶セグメントにアクセスすることを可能とするために必要である。即ち、呼び出しプロセス P が O に対して直接カーネル操作を行うことができなくとも、M がそれを行うとことを可能とする。O に対する操作 r が完了した後、O のケーパビリティ C_a は元の状態、即ち、非増幅状態 C に戻される。これは、実行される仕事の状態に応じて、保護されたセグメントにアクセスするプロセスによって保持される権利は動的に変更されなければならないという典型的な場合である。

権利増幅方式では、オブジェクト管理者は「信頼できる」手続きとして扱われ、特定の型のオブジェクトに対するカーネル操作を行うことが可能である。これは、その型のオブジェクトの補助権利を有する任意のプロセスに代わって行う。したがって、オブジェクト管理者が保持する権利は、そのオブジェクトにアクセスするサブジェクトが保持する権利と独立であるか、又は、通常はそ

れを超えるものである。しかし、オブジェクト管理者は普遍的に信頼できる手続きというものではない。それは、オブジェクト管理者は他の型のオブジェクトには作用できず、他の任意の手続きに権利を拡張することもできないからである。

権利取り消し

ACL に基づくセキュリティシステムでは、**権利取り消し (rights revocation)** は簡単である。これは、与えられたオブジェクトに対して、どのサブジェクトがどのような権利を有するかを、簡単にかつ効率良く決定することができるからである。しかし、ケーパビリティに基づくセキュリティシステムでは、権利取り消しははるかに難しい問題となる。与えられたオブジェクトに対して、どのサブジェクトがどのような権利を有するかを決定することが難しいからである。あるオブジェクトに関するケーパビリティがシステム全体に分散した複数のケーパビリティリストに格納されているかもしれず、それらを取り消す前に、まずそれらを見つけ出さなければならない。以下では、ケーパビリティの取り消しを実装するのに通常用いられている方法を説明する。

1. **後方ポインタ** (back pointer):1 つの方法は、あるオブジェクトに関するすべてのケーパビリティを追跡し、要求されている取り消しに依存して、選択的にケーパビリティを変更/削除することである。あるオブジェクトに関するすべてのケーパビリティを追跡する簡単な方法は、そのオブジェクトとともにポインタリストを維持管理することである。これらのポインタはそのオブジェクトに付随するすべてのケーパビリティを指している。この方法は Multics システムで用いられた。この方法は極めて一般的であるが、実装するには非常に高価である。

2. **間接化** (indirection):もう 1 つの方法は、間接アドレッシングを用いるものである。この方法では、各ケーパビリティはオブジェクトそのものではなく、間接的なオブジェクト (表要素など) を指す。代わりに、間接オブジェクトが本当のオブジェクトを指す。取り消しは、間接オブジェクトを削除することによって、本当のオブジェクトとそのケーパビリティの間の接続を破壊することによって実現される。間接オブジェクトが削除されたケーパビリティへのアクセスを試みた場合、接続が破壊によって本当のオブジェクトを知ることができないので、そのアクセス操作は失敗する。

3. 鍵の使用 (use of key):この方法では、オブジェクト識別子欄と権利情報欄に加えて、各ケーパビリティはユニークなビットパターンを含む欄を持つ。この欄の内容は「鍵」と呼ばれる。ケーパビリティの鍵は、そのケーパビリティが生成された時に定義され、そのケーパビリティを所有するプロセスが修正したり、検査したりすることはできない。各オブジェクトは付随する**マスター鍵** (master key) を持つ。その鍵は、特別の set_key 操作によって動的に定義したり、変更したりできる。通常は、オブジェクトの所有者のみが、そのオブジェクトのマスター鍵を変更するための set_key 操作を起動する権利を与えられる。

あるオブジェクトに対して新しいケーパビリティが生成された場合、そのケーパビリティの鍵欄はそのオブジェクトのその時点のマスター鍵に設定される。あるケーパビリティを持ってアクセスが試みられた場合、そのケーパビリティの鍵が、対応するオブジェクトのマスター鍵と比較される。この 2 つが同じであれば、そのオブジェクトへのアクセスが許可される。同じでなければ、保護違反が発生する。取り消しは、set_key 操作を用いてマスター鍵を新しい値で置きかえることによって行われる。その結果、このオブジェクトに関する以前のすべてのケーパビリティは無効化される。

この取り消し方式は、Amoeba で用いられ、そこでは鍵として乱数が用いられた。この方式の欠点は、マスター鍵のみが各オブジェクトに付随しているため、選択的な取り消しができないことである。しかし、この欠点は、各オブジェクトに鍵のリストを付随させることで克服できる。

複合方式

ケーパビリティに基づく方式と比較して、ACL に基づく方式の方がセキュリティシステムの実装に適している。ACL がユーザの要求に直接対応しているからである。ユーザがオブジェクトを生成した場合、ユーザは許される操作とともにどのドメインがそのオブジェクトにアクセスできるかを指定することができる。しかし、ACL に基づくセキュリティシステムは、各アクセスごとにアクセスリストを探索することから、ケーパビリティに基づくセキュリティシステムに比べて効率が悪いことを見てきた。これら 2 つの方式の欠点を克服し、これらの利点を結合するために、大部分のシステムはセキュリティシステムの設計に複合方式を用いている。

複合方式では、セッションの概念とともに、ACL とケーパビリティの両方が用いられる。セッション (session) とは、プロセスがオブジェクトをアクセス

する期間の論理的な概念である。プロセスがあるオブジェクトのアクセスに関するセッションを開始した時に、そのセッション中にオブジェクトに対して行われる操作のアクセスモード (操作の型) を指定する。そのオブジェクトのACLが探索され、要求された操作の型を見出す。アクセスが拒否された場合には、保護違反が発生する。さもなければ、システムはそのオブジェクトに関する新しいオブジェクトを生成し、それを当該プロセスに付加する。その後、セッション中にこのプロセスによってそのオブジェクトに対して行われるすべてのアクセスは、そのケーパビリティを用いて行われる。その結果、アクセス制御確認を効率良く行うことができる。セッションが終了した後には、ケーパビリティは破壊される。

UNIX ファイルシステムはこの方式を用いている。ここで、あるプロセスによる、ファイルの open 操作と close 操作の間の期間がセッションである。各ファイルは付随する ACL を持つ。プロセスがファイルをオープンした時、open コマンドに指定されたモードに対して ACL が確認される。もし指定されたモードが許可されていれば、ファイル表の中に新しいエントリが割り当てられ、そのアクセスモードがそのエントリに記録され、このエントリに対するインデックスがプロセスに返される。このプロセスによって行われる、そのファイルに対して引き続くすべての操作は、ファイル表へのインデックスを指定することによって行われる。ファイル表のエントリはファイルを指す。プロセスがそのファイルに対して close 操作を行った場合には、ファイル表のエントリを削除することによって、セッションが閉じられる。そのプロセスが同じファイルを後でアクセスしたい場合には、新しいセッションは、close 操作の後で開始されなければならない。

ファイル表はオペレーティングシステムによって維持管理される。そのため、ユーザによって汚染される可能性はない。アクセスがセッション開始時に正当化され、プロセスはセッションが開始され、未だに閉じられていないファイルに対してのみアクセス可能であることから、セキュリティが保証される。

11.6　デジタル署名

メッセージの完全性は、転送時にメッセージの内容が変更されていないことを保証するもので、分散システムにおける重要なセキュリティ要求でもある。デジタル署名の概念は、非対称暗号システムに基づいており、このメッセージの完全性の問題を扱う最も一般的な方法である。

非対称暗号システムでは、あるユーザの秘密鍵はそのユーザだけが知っており、他のユーザには未知である。したがって、メッセージの送信者は、メッセージをその鍵で暗号化することによって、秘密鍵をメッセージへの署名に用いることができる。即ち、送信者は自分の署名 (秘密鍵) で、メッセージをユニークに「封印」することができる。封印されたメッセージは、対応する公開鍵を用いて、任意のユーザに送信することができる。デジタル署名を用いることによって、メッセージの受信者に対して、メッセージに内容が操作されていないことと、そのメッセージが主張通りの送信者によって送信されたものであることを保証できる。したがって、デジタル署名は、ユーザ認証とメッセージの完全性の両方に適用可能なものである。

デジタル署名 (digital signature) は基本的に符号又は大きな数字であり、各メッセージと各メッセージ送信者に関して一意である。デジタル署名は、まずメッセージをハッシュ関数 (**要約関数** (digest function) と呼ばれる) で処理して、メッセージ内の情報の各ビットに依存した小さな要約を作り、次にその要約を、送信者の秘密鍵で暗号化することによって得られる。重複の問題を回避するために、要約関数 (D) は、可能なメッセージの組 M、M' に関して、$D(M)$ と $D(M')$ が異なるという特性を持たなければ成らない。Rivest[1992] は、安全なメールなどのインターネットアプリケーション向けのメッセージ要約関数 ($MD5$ として知られている) を提案している。

どのようにしてメッセージから要約メッセージが得られるかを説明するために、ここでは [Adam 1992] で述べられた例を示す。この例では、メッセージは 0 と 1 から成るデジタル列であり、64 ビットのブロックに分割されていると仮定する。最初の 2 つのブロックに対して、ビットごとの排他的論理和をとり、新しい 64 ビットブロックを得る。この新しく得られたブロックと 3 番目のブロックとのビットごとの排他的論理和をとり、再び新しい 64 ビットブロックを得る。このプロセスを他のメッセージブロックに対して繰り返す。この結果、得られる 64 ビットブロックは、メッセージ全体のデータの各ビットに依存した 64 ビット要約である。言い換えれば、たとえ 1 ビットでもメッセージを変更すると、64 ビット要約も変わってしまう。さらに、同じ要約となるようなメッセージを偽造することも本質的に不可能である。

メッセージの完全性を保証するための、デジタル署名に基づくプロトコルは以下のように動作する。

1. 送信者 (A) はメッセージ (M) の要約 (D) を計算する。次に、秘密鍵 (S_a)

で要約 D を暗号化し、暗号文 $C_1 = E(D, S_a)$ を得る。送信者の識別子、平文形式のメッセージ M 及び暗号文 C_1 から署名されたメッセージを作る。この署名されたメッセージは (ID_a, C_1, M) の形を持ち、受信者へと送られる。

2. 署名されたメッセージを受信すると、受信者は送信者の公開鍵を用いて C_1 を復号し、要約 D を回復する。次に、(同じ要約関数を用いて)M の要約を計算し、それと C_1 から回復された要約とを比較する。これらが等しければ、メッセージ M は正しいと考えられ、そうでなければ、不正と考えられる。

このプロトコルでは、メッセージを許可されていないユーザから隠す必要がないことに注意が必要である。さらに、メッセージは、受信又は盗聴したすべての人に読まれてしまう。しかし、偽造されたメッセージはプロトコルによってうまく検出される。

アプリケーションが、受信された署名メッセージを次から次へと転送することを要求するかもしれない。このような状況では、署名メッセージが本当の送信者から送られたものであり、またその内容が中間の受信者や侵入者によって改ざんされていないことを、各受信者が確認できることが重要である。デジタル的に署名されたメッセージはこれらの要求に合致するものである。それは、送信元の識別子が含まれているし、メッセージの要約は送信元の公開鍵を用いてのみ復号できるからである。

実際の実装では、すべてのユーザの公開鍵データベースを管理する鍵配布サーバが用いられる場合もある。デジタル的に署名されたメッセージの受信者がメッセージ送信元の公開鍵を持っていない場合には、それを鍵配布サーバに要求する。これによって、新しいユーザの公開鍵をシステム内の他のすべてのユーザに送る必要をなくすことができる。新しいユーザの公開鍵は、単に鍵配布サーバに登録するだけでよい。

プライバシ拡張メール (Privacy Enhanced Mail)(PEM) 方式は、インターネットメールアプリケーションにプライバシを追加するために設計されたものであり、暗号とデジタル署名技術の有効活用例である。PEM では、秘匿性、認証及びメッセージの完全性が提供される。これらの特徴は、十分な信頼性を提供して、一般のインターネットユーザがビジネス文書や重要な情報を含んだメッセージの送信を安心して行えるようにすることを意図している。PEM は、エンドシステムのアプリケーションレベルで実装され、その結果、サイト単位やユーザ単位で導入することができる。この方式は、中継点や端点における

メッセージ転送システムに特別の要求を課さない。即ち、ネットワークルータやメール中継器は、PEM メッセージを通常のメール断片として扱う。PEM がどのようにして電子メールにプライバシを提供するかについては、以下で簡単に説明する。詳細な説明に関しては、[Lin 1993, Kent 1993b, Balenson 1993, Kaliski 1993, Kent 1993a] を参照のこと。

PEM では、ネットワークは信頼できないが、PEM のユーザは自身のコンピュータを信頼していることを仮定する。メールユーザはローカルな PEM プログラムから公開鍵と秘密鍵の組を獲得し、公開鍵は自身のメールアドレスとともに公表する。PEM プログラムは、ローカルユーザの秘密鍵と遠隔ユーザの公開鍵のデータベースを管理する。現時点では、ユーザの公開鍵と秘密鍵の組を生成するのに、Rivest-Shamir-Adleman のアルゴリズムが用いられている。PEM は以下のタイプの機能を提供する。

1. **秘匿性** (confidentiality):重要な情報が侵入者に読まれないように、メッセージを暗号化された形式で送る。

2. **メッセージの完全性** (message integrity):受信者に対してメッセージの内容が変更されていないことを保証するように、署名されたメッセージを送る。

これらの機能は認証機能も持つ。メッセージの暗号化/復号化やデジタル署名は、適当な鍵を持つユーザだけが実行できるからである。

まず、秘匿性の保証のために、PEM がどのように秘密メッセージ (M) を送るかについて説明する。

1. まず、送信側のコンピュータの PEM プログラムがランダムな秘密鍵 (K) を生成し、この鍵を用いてメッセージ (M) を暗号化して、暗号文 $C_1 = E(M, K)$ を得る。現時点では、この目的のために DES アルゴリズムが用いられているが、将来には他のアルゴリズムが用いられる可能性もある。秘密鍵 (K) は、受信側の公開鍵 (P_r) を用いて暗号化され、暗号文 $C_2 = E(K, P_r)$ を得る。この C_1 と C_2 がメッセージ m_1 に含まれて、受信側に送られる。

2. メッセージ m_1 を受信すると、受信側コンピュータの PEM プログラムは、そのデータベースから受信側の公開鍵 (S_r) を読み出し、S_r を用いて C_2 を復号して K を得る。次に、K を用いて、C_1 を復号し、元のメッセージ m_1 を得

る。この M が受信側のメールボックスに格納される。

PEM 方式は大規模な暗号化に関して、対称暗号システムの効率を維持しているが、安全な鍵配布サーバを不要としている点に注意する必要がある。

ここで、メッセージの完全性のために、PEM がどのように署名されたメッセージを送るかについて説明する。

1. 送信側コンピュータの PEM プログラムは、メッセージ要約関数を用いて、メッセージ (M) の要約 (D) を計算する。次に、送信者の秘密鍵 (S_s) を用いて要約 (D) を暗号化し、暗号文 $C_1 = E(D, S_s)$ を得る。送信者の ID、C_1 および M を、メッセージ m_1 に含めて受信側に送る。

2. メッセージ m_1 を受信すると、受信側コンピュータの PEM プログラムは、そのデータベースから送信者の秘密鍵 (P_s) を読み出し、それを用いて C_1 を復号し、要約 D を得る。次に、同じ要約関数を M に適用して、その結果を D と比較する。これらが等しければ、メッセージ M は正しいと考えられる。さもなければ、正しくないと考えられる。メッセージ M は、完全性確認に結果に関する PEM としてのコメントを付加されて、受信側のメールボックスに格納される。

11.7　設計原則

Multics の経験に基づいて、Saltzer と Schroeder[1975] は、安全なシステムの設計のガイドとして用いることができる、幾つかの設計原則を特定している。これらの設計原則は集中型システムに関して提案されたものであるが、分散システムにも有効である [Kent 1981]。これらを含む幾つかの設計原則を以下に要約する。分散オペレーティングシステムのセキュリティ構成要素の設計者はこれらを基礎的なガイドラインとして用いるべきである。

1. **最小特権** (least privilege): 最小特権原則 (必要性原則としても知られている) とは、プロセスには、任意の時点でその機能を果たすのに必要不可欠なものに対するアクセス権のみを与えるべきであるという考え方である。即ち、セキュリティシステムは、アクセス要求の変化に応じて、プロセスのアクセス権を拡大したり、縮小したりできるような柔軟性を持たなければならない。この

原則は、セキュリティシステムが破壊された場合の損害を限定するのに役立つ。例えば、エディタに、エディットすべきファイルのみへのアクセス権が与えられていれば、たとえそのエディタがトロイの馬を持っていたとしても、ユーザの他のファイルにはアクセスできず、その結果、それほど大きな損害は生じないことになる。

2. **既定値によるフェイルセーフ:** アクセス権は、明確な許可によって獲得されるべきであり、その既定値はアクセス権なしとすべきである。この原則によって、アクセス制御の決定は、なぜあるオブジェクトがあるプロセスにとってアクセス不可能であるかという観点ではなく、なぜアクセス可能かという観点から行われる必要がある。

3. **開放型の設計:** この原則によって、セキュリティ機構の設計は秘密ではなく、公開されることが必要となる。侵入者はセキュリティ機構がどのように動作するかを知ることはないという仮定は、設計者側の誤りである。

4. **システムへの組み込み:** この原則によって、セキュリティは最初からシステムに組み込まれるよう設計すべきであり、しかもシステムの最下層に組み込むべきである。即ち、セキュリティは付加的な特徴として扱うべきではない。セキュリティ問題は、既存システムで発見されたセキュリティホールにパッチを当てることでは、効果的に解決できないからである。

5. **その時点の権限の確認:** この原則によって、各オブジェクトに対する各アクセスが、権限に関するアクセス制御データベースを用いて確認されなければならない。以前に与えられたアクセス権の取り消しが即時に効果を持つことが必要である。例えば、幾つかのファイルシステムでは、アクセス許可の確認はファイルをオープンした時にのみ確認され、そのファイルへのそれ以降のアクセスは確認なしに行われる。このようなシステムでは、ユーザがファイルを数日間オープンしたままにした場合、たとえそのファイルの所有者がアクセス許可を変更し、ユーザのアクセス権を取り消しても、アクセスを継続することができる。

6. **アクセス権の容易な承認と取り消し:** 柔軟性を高めるために、セキュリティシステムはオブジェクトへのアクセス権が動的に承認されたり、取り消さ

れたりすることを可能としなければならない。権利を制限することが可能であるべきであり、機能を果たすために十分な機能のみがユーザに対して承認されるべきである。一方、優れたセキュリティシステムでは、選択的取り消しや部分取り消しを伴う、即時取り消しを可能とすべきである。**選択的取り消し** (selective revocation) 機能によって、あるオブジェクトに対するアクセス権を持つすべてのユーザではなく、選択されたユーザグループからのそのオブジェクトに対するアクセス権を取り消すことが可能となる。また、**部分取り消し** (partial revocation) 機能によって、あるオブジェクトに関する権利をすべて取り消すのではなく、そのオブジェクトに関してユーザに承認された部分的な権利のみを取り消すことができる。

7. **他の当事者の不信**: 安全な分散システムを構築するためには、他の当事者 (人やプログラム) は、それらが信頼できるということが証明されるまでは、信頼できないという前提を持って、そのシステムの構成要素を設計しなければならない。例えば、クライアントとサーバは、お互いを疑いの目で見るように設計されなければならない。

8. **メッセージの最新性の保証**: メッセージの再送によるセキュリティ違反を防ぐために、分散システムのセキュリティは、2 つの通信実体間で交換されるメッセージの最新性を常に保証するよう設計されなければならない。

9. **ファイアウォールの構築**: システムにセキュリティが破られた場合の損害を限定するために、システムにはファイアウォールが組み込まれていなければならない。この要求を満足する 1 つの方法は、システムにおいて有効期間の短いパスワードや鍵の使用のみを可能とすることである。例えば、クライアントとサーバの間に論理的な通信チャネルを構築するのに用いられる共有秘密鍵は、その有効期間をかなり短くすべきであり、大抵の場合、各通信セッションごとに変更される。

10. **効率性**: 用いられるセキュリティ機構は、効率良く実行できなければならず、また実装が簡単でなければならない。

11. **使用の容易性**: 心理的に受け入れ可能にするために、セキュリティ機構は容易に使用できなければならない。さもなければ、ユーザによって回避され

たり、間違った使い方をされてしまう。

12. **コスト効率性:** セキュリティと性能や使用の容易性などのシステム目標とのトレードオフを考える必要性があることはよく見られることである。したがって、システムのセキュリティ設計においては、セキュリティを提供することによるコスト (価格と人の習熟度) と妥協しなければならないということを考慮した、適切なトレードオフの集合を考え出すことが重要である。

11.8 事例研究：DCE セキュリティサービス

本章で述べた各種のセキュリティ概念が、1つのシステムのセキュリティを提供するために、どのように統合されるかの事例研究として、以下では、DCE セキュリティサービスを簡単に説明する。

DCE では、安全に通信することが必要なユーザやプロセス (クライアント又はサーバ) は、**プリンシパル** (principal) と呼ばれる。アクセス制御のために、プリンシパルには1つ以上の**グループ** (group) 及び**組織** (organization) へのメンバーシップが与えられる。同じグループ又は組織のプリンシパルは、同じアクセス権を持つ。一般に、グループは仕事上のグループや部門に相当し、組織は何らかの共通の特性を持った複数のグループから構成される。典型的には、プリンシパルは1つの組織のメンバーであるが、同時に複数のグループのメンバーであってもよい。各プリンシパルはユニークな識別子を持つ。プリンシパルの識別子、グループへのメンバーシップ及び組織へのメンバーシップは、まとめてプリンシパルの**特権属性** (privilege attribute) として知られている。

単一セルに関する DCE セキュリティシステムの主要構成要素を図 11.21 に示す。これらの構成要素は、それぞれ認証サービス、認定サービス、メッセージの完全性サービス及びセキュリティ管理サービスを提供する。以下では、これらのサービスを1つ1つ説明していく。

11.8.1 DCE における認証

DCE 認証サービスは、11.4.5 で述べたケルベロスシステムを使用している。ケルベロスの認証サーバ、チケット承認サーバ、及び認証データベースは、DCE ではそれぞれ認証サーバ、特権サーバ及びレジストリサーバと呼ばれる。レジストリデータベースに登録される情報には、各プリンシパルの秘密鍵と特

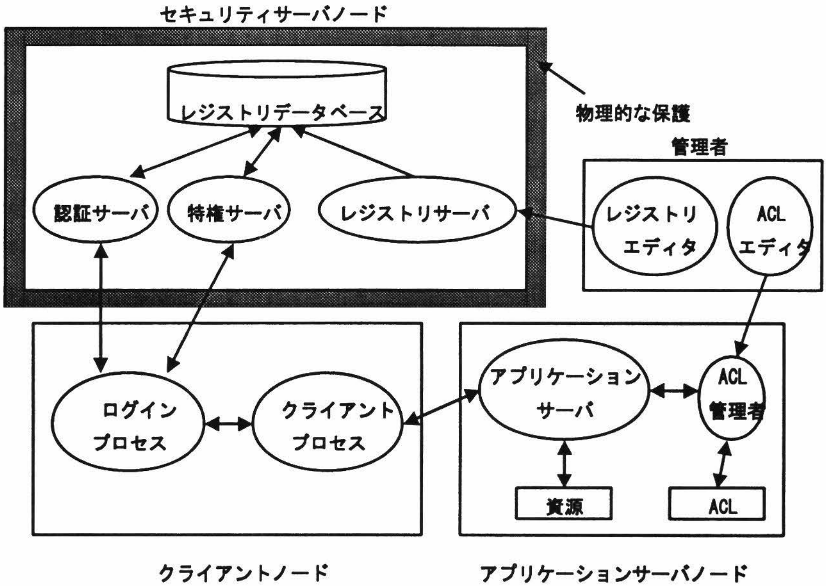


図 11.21　1 つのセルに関する DCE セキュリティサービスの主要構成要素

権属性が含まれている。ログイン時のユーザ認証プロトコルと、クライアントとサーバの相互認証プロトコルは、ケルベロスのそれと同じである (DCE のセルをまたがるクライアントサーバ間認証プロトコルは、ケルベロスにおけるレルム間認証プロトコルと同じである)。唯一の違いは、DCE のサービス承認チケットは、クライアントのグループ及び組織のメンバーシップ情報も含むことである。この情報は、アプリケーションサーバによって使用され、要求されたサービスを実行する前に、クライアントのアクセス権が確認される。

認証プロトコルを用いた、クライアントとサーバの間の安全な論理的通信チャネルの確立は、DCE では**認証済み RPC** (authenticated RPC) として知られている。これは、DCE ではクライアントとサーバが RPC を用いて通信するからである。一旦認証済み RPC が確立されると、どの程度のセキュリティを必要とするかを決定するのはクライアントとサーバの責任である。即ち、それ以降の RPC メッセージは、アプリケーションのセキュリティ要求に応じて、暗号化してもしなくてもよい。

11.8.2 DCEにおける認定

DCE における認定は ACL に基づいている。各アプリケーションサーバには ACL と ACL 管理者が付随している。ACL には、サーバが管理している資源に関してプリンシパルがどのような権利を持つかについての一式の情報が含まれている。クライアントの要求がサーバに届いた時、サーバは受信した暗号化されたチケットから、クライアントの ID、グループメンバーシップ及び組織メンバーシップを取り出す。次に、クライアントの ID、メンバーシップ及び要求された操作を ACL 管理者に渡す。この情報を用いて、ACL 管理者は ACL を確認し、そのクライアントの要求の実行が認定されるか否かを決定する。その結果、アクセス承認又は拒否の応答を返し、その後、サーバはそれに従って動作する。

11.8.3 DCEにおけるメッセージの完全性

上述のように、一旦認証済み RPC が確立されれば、どの程度のセキュリティを必要とするかを決定するのはクライアントとサーバの責任である。したがって、メッセージの完全性が必要であれば、デジタル署名技術を用いてそれを保証することができる。即ち、アプリケーションは、クライアントとサーバ間のメッセージデータの暗号化された要約を含めることによって、データの完全性を保証することができる。要約は、安全な通信のためにクライアントとサーバが共有するセッション鍵を用いて、暗号化／復号化されなければならない。

11.8.4 DCEにおけるセキュリティ管理

管理者、レジストリサーバ及び ACL 管理者が協力してセキュリティ管理業務を行う。管理業務を行うために、管理者は 2 つのプログラムを用いる。レジストリエディタプログラムと ACL エディタプログラムである。

システム管理者は、レジストリエディタプログラムを用いて、レジストリデータベースの情報を、見たり、追加したり、削除したり、修正したりする。システム管理者でさえ、レジストリデータベースには直接アクセスできず、レジストリサーバへ要求することによってのみアクセスすることができる。これは非常に安全な方法である。管理者は任意のパスワードを変更することができるが、任意のユーザのパスワードを獲得することができないからである。

一方、ACL エディタプログラムを用いて、アプリケーション管理者は、アプリケーションの ACL 又はそれらによって制御されているオブジェクト (資

源) の ACL のエントリを見たり、追加したり、削除したり、修正したりする。ACL の更新にかんするすべての要求は ACL 管理者に送られ、ACL に対して直接行われることはない。

DCE では、大域的なセキュリティ方針を適用するために、組織のメンバーシップが用いられる。大域的なセキュリティ方針とは、異なるプリンシパルのチケットやパスワードの有効期間を定めたりすることである。例えば、重要な情報を扱う組織のプリンシパルのチケットやパスワードの有効期間は、そうではない組織のプリンシパルのそれに比べて、短く設定される。

11.9　まとめ

コンピュータセキュリティは、コンピュータシステム内の各種資源や情報を、破壊や許可されていないアクセスから保護する問題を扱う。コンピュータセキュリティの主要目標は、安全性、プライバシ、信憑性及び完全性である。

コンピュータセキュリティ全体は、外部セキュリティと内部セキュリティから構成される。分散システムにおける内部セキュリティの主要な 3 つの側面は、認証、アクセス制御及び通信の安全性である。

侵入者とは、コンピュータシステムのデータ又は資源に許可されていないアクセスを行おうとする人又はプログラムである。侵入者は、多くの点でコンピュータセキュリティに対する脅威となり、大きく分けて 2 つの範疇に分類される。即ち、受動的攻撃と能動的攻撃である。受動的攻撃では、侵入者は、システムの通常の動作に干渉することなく、コンピュータシステムから許可されていない情報を盗もうとする。よく用いられる受動的攻撃は、ブラウジング、漏洩、推定及び偽りである。一方、能動的攻撃はシステムの通常の動作に干渉し、損害を与えるのが普通である。よく用いられる能動的攻撃は、ウィルス、ウォーム及び論理爆弾である。メッセージ通信に伴う能動的攻撃には、完全性攻撃、信憑性攻撃、拒絶攻撃、遅延攻撃及び再送攻撃がある。

プログラムが情報を漏洩するために用いる 3 つのチャネルは、合法的チャネル、記憶チャネル及び秘密チャネルである。監禁問題とは、許可されたサブジェクトがアクセス権を持つオブジェクトの情報を、他の許可されていないサブジェクトに対してリリースするすべての方法を除去する問題である。一般的に、監禁問題は解決不可能である。

暗号は、物理的な保護手段を講じるのが困難な状況で、許可されていないアクセスから私的な情報を保護する方法である。暗号システムには 2 つの大きな

クラスがある。即ち、対称暗号と非対称暗号である。分散システムにおける安全な通信に暗号が用いられる場合、鍵配布の問題が発生する。本章では、対称暗号システムと非対称暗号システムにおける鍵配布の機構とプロトコルについて説明した。

認証機構は、要求を行うユーザの身元を確認することによって、許可されていないユーザがシステム(又はシステムの何らかの資源)を使用することを禁止する。通常、分散システムで必要とされる認証のタイプは、ユーザログイン認証、通信実体の一方向認証及び双方向認証である。認証に関する3つの基本的な方法は、知識による証明、所有による証明及び特性による証明である。パスワードに基づいた知識による証明法は、ユーザログイン認証で最も広く用いられている方法である。通信実体の一方向認証と双方向認証に関しては、本章では暗号システムに基づいたプロトコルを説明した。事例研究として、ケルベロス認証システムについても説明した。

アクセス制御は、コンピュータシステムにおいて、アクセスを許可されていないユーザが資源や情報にアクセスすることを禁止する方法を扱うものである。文献で提案されている3つのアクセス制御モデルは、アクセス行列モデル、情報フローモデル及びセキュリティカーネルモデルである。これらの中で、アクセス行列モデルが最もよく知られており、既存の集中型システムや分散システムで広く用いられている。

アクセス行列モデルでは、各サブジェクトの各オブジェクトに対するアクセス権は、アクセス行列と呼ばれる行列の要素として定義される。アクセス行列の実装に関して、現在の分散システムでよく知られるようになり、最も広く用いられている方法は、ACLとケーパビリティである。

デジタル署名の概念は、非対称暗号システムに基づいており、分散システムにおけるメッセージの完全性問題を扱うのに、最も普及しているものである。

安全なシステムを設計するガイドとして用いることができる設計原則には、最小特権、既定値によるフェイルセーフ、開放型の設計、システムへの組み込み、その時点の権限の確認、アクセス権の容易な承認と取り消し、他の当事者に不信、メッセージの最新性の保証、ファイアウォールの構築、効率性、使用の容易性及び適切なトレードオフ集合がある。

11.10 演習問題

11.1 コンピュータセキュリティの共通目標を列挙せよ。

11.2 集中型のタイムシェアリングオペレーティングシステムの設計者に比べて、分散オペレーティングシステムの設計者が対処しなければならない、付加的なセキュリティ問題とは何か。集中型のタイムシェアリングシステムと同等のセキュリティを分散システムでも実現することは可能か。理由とともに解答せよ。

11.3 コンピュータセキュリティにおける「必要性原則」とは何か。コンピュータシステムにおけるセキュリティ構成要素を設計するに当たって、この原則を考慮しなければ、どのようなセキュリティ問題が発生するかを検討せよ。

11.4 受動的攻撃と能動的攻撃の違いを述べよ。いずれが重大な問題か、またその理由は何か。

11.5 受動的攻撃としてよく用いられる方法にはどのようなものがあるか。以下の観点から、それらの方法の相対的な複雑さについて述べよ。

(a) 侵入者

(b) セキュリティシステムの設計者

11.6 トロイの馬プログラムとは何か。受動的なトロイの馬プログラムと能動的なそれの例 (擬似コード) を示せ。

11.7 ウィルス感染からコンピュータシステムを守るために必要な予防策を列挙せよ。既にウィルスに感染したコンピュータシステムはどのようにして治療可能か。ウィルスに感染した分散システムの治療が、集中型システムの治療よりも困難な理由は何か。

11.8 コンピュータウィルスとウォームの重要な相違点を列挙せよ。それぞれは、どのようにして複製を作るか。あるセキュリティシステムが、ウィルスプログラムが制御できない形で複製を作のを防ぎながら、ウォームプログラムを実行可能とするように設計されるとする。これに適したセキュリティ方式を提案せよ。

11.9 分散システムにおいてよく見られる、メッセージ通信に付随する能動的攻撃のタイプは何か。以下の観点から、それらのタイプの相対的な複雑さについて述べよ。

(a) 侵入者

(b) 分散システムにおけるセキュリティシステムの設計者

11.10 ノンスとは何か。ノンスとして使用可能なものの例を挙げよ。

11.11 コンピュータセキュリティにおける監禁問題とは何か。この問題が一般的には解決不可能な理由を述べよ。

11.12 プログラムが情報漏洩のために用いることができるチャネルのタイプ

を列挙せよ。情報漏洩のために、これらのチャネルがどのように用いられるかを説明せよ。情報の漏洩を完全に防ぐことは可能か。

11.13 データオブジェクトのみの共有を可能とするシステムのセキュリティと、データオブジェクトとプログラムオブジェクトの双方の共有を可能とするシステムのセキュリティとの、設計上の相対的な複雑さについて説明せよ。

11.14 暗号とは何か。分散システムにおいて、よく見られる用途は何か。

11.15 優れた暗号システムが満足しなければならない基本的な要求は何か。

11.16 暗号システムに関して、暗号文攻撃、既知平文攻撃及び選択平文攻撃の違いを述べよ。

11.17 対称暗号システムと非対称暗号システムに動作を説明せよ。これらの相対的な長所、短所を説明せよ。以下の場合に、2つの内、どちらが適しているか(その理由も示せ)。

(a) 情報の暗号化と復号化が共に信頼できるサブシステムで行われる場合

(b) 情報の暗号化と復号化を行うサブジェクトが異なる場合

(c) 大量データの暗号化が含まれる場合

(d) 分散システムにおいて、2つの通信実体間の接続を確立する場合

(e) 分散システムにおいて、2つの通信実体間でメッセージの交換を行う場合

11.18 鍵配布問題とは何か。この問題は、対称暗号システムと非対称暗号システムでどのように異なるか。

11.19 対称暗号システムにおける鍵配布問題の2つの解決策について述べよ。また、それらの相対的な長所、短所を説明せよ。

11.20 非対称暗号システムにおける鍵配布問題の解決策について述べよ。

11.21 コンピュータシステムにおけるユーザ認証方法として、よく用いられるものは何か。それぞれの方法において、どのようにユーザが認証されるかを説明せよ。

11.22 ユーザログイン認証に関するパスワードに基づく方法を説明せよ。この方法に伴う問題は何か。それらの問題を克服する方法を提案せよ。

11.23 分散システムでは、サーバは、クライアントの身元を確認するだけで、そのサービスを必要とする任意のクライアントにサービスを提供することが望ましい。認証に関するこの要求を実現する方法を述べよ。

11.24 前問において、通信セッションを開始する前に、クライアントとサーバの双方がお互いに身元を確認すると仮定する。認証に関するこの要求を実現する方法を述べよ。

11.25 ログイン時にユーザを認証するために、分散システムにおいてパス

ワードに基づく機構が用いられるとする。以下の場合について、ログインプログラムとパスワードファイルを格納するのに最も相応しい場所 (開発者にとって最も相応しい場所) を述べよ。

(a) その分散システムがワークステーション-サーバモデルであり、各ワークステーションが約 20 MB の小容量のハードディスクしか持たない場合

(b) その分散システムがワークステーション-サーバモデルであり、幾つかのワークステーションにはディスクレスで、その他は約 20 MB の小容量のハードディスクしか持たない場合

(c) その分散システムがプロセッサプールモデルである場合

11.26 アクセス行列とは何か。アクセス制御にアクセス行列を用いるようなセキュリティシステムにおいて、以下の問題がどのように扱われるかを説明せよ。

(a) アクセス行列要素の内容の決定

(b) サブジェクトによるオブジェクトアクセスの正当化

(c) 制御された形でのサブジェクトのドメイン遷移の可能化

11.27 アクセス行列を実現するのによく用いられる方法は何か。それらの相対的な長所、短所を述べよ。

11.28 ドメインとは何か。オペレーティングシステムのセキュリティ構成要素の設計において、ドメインを実現する 3 つの方法を述べよ。また、それぞれの場合に、ドメイン遷移がどのようにして行われるかを説明せよ。UNIX と Multics において、ドメインがどのように形成されるか説明し、またどのようにドメイン遷移が起こるかを説明せよ。

11.29 ケーパビリティとは何か。ケーパビリティに基づくセキュリティシステムに関して、以下の設問に解答せよ。

(a) あるサブジェクトがあるオブジェクトにアクセスする場合、要求されたモードでのそのオブジェクトへのアクセスが許されるか否かの正当性確認はどのように行われるか

(b) サブジェクトはオブジェクトのケーパビリティをどのようにして獲得するか。そのサブジェクトがそのオブジェクトの所有者である場合と、所有者でない場合の両方を検討せよ。

(c) オブジェクトの所有者が、他のサブジェクトとそのオブジェクトを共有するが、そのアクセス権を制限したいとする。どのようにすればこれが可能か。

(d) そのシステムにおいて、あるケーパビリティが再利用され、他のオブジェクトを特定することがないことを保証できるようにするにはどうするか。

(e) どのようにすれば、ケーパビリティを許可されていないアクセスから保

護できるか

(f) どのようにすれば、ケーパビリティを推測困難にできるか

(g) どのようにすれば、ケーパビリティを偽造困難にできるか

(h) どのようにすれば、ケーパビリティの選択的取り消しを行えるか

11.30 ACL に基づくセキュリティシステムに関して、以下の設問に解答せよ。

(a) あるサブジェクトがあるオブジェクトにアクセスする場合、要求されたモードでのそのオブジェクトへのアクセスが許されるか否かの正当性確認はどのように行われるか

(b) オブジェクトのアクセス権は、どのようにサブジェクトに対して承認されるか

(c) オブジェクトの所有者が、他のサブジェクトとそのオブジェクトを共有するが、そのアクセス権を制限したいとする。どのようにすればこれが可能か。

(d) どのようにすれば、ユーザによる変更から ACL を保護することができるか

(e) どのようにすれば、サブジェクトに与えられたアクセス権を取り消すことができるか

11.31 分散システムにおいては、アクセス要求が行われるたびに、サーバがクライアントのアクセス権を確認することが望ましい。以下の手法を用いた場合、これはどのように実現することが可能か。

(a)ACL のみ

(b) ケーパビリティのみ

(c)ACL とケーパビリティの両方

どの方法が好ましいか。また、その理由は何か。

11.32 ある分散オペレーティングシステムでは、ACL に基づいたアクセス制御機構を用いているが、否定権利を用いていないとする。このシステムでは、どのタイプのアクセス制御を行うのが困難か。ACL に基づいたアクセス制御機構を用いた集中型システムではこの問題に直面しないのは何故か。

11.33 あるシステムが多数のユーザを持つとする。このシステムでは、ファイルを、5 人のユーザを除くすべてのユーザからアクセス可能とすることが必要である。以下のそれぞれの場合において、このセキュリティ要求はどのように記述されるか。

(a) 否定権利機能を持たない ACL に基づいたセキュリティ方式の場合

(b) 否定権利機能を持つ ACL に基づいたセキュリティ方式の場合

(c) ケーパビリティに基づいたセキュリティ方式の場合

11.34 デジタル署名とは何か。分散システムにおける用途は何か。デジタル署名を生成する方法の例を挙げよ。分散システムにおいて、メッセージの完全性のためにデジタル署名がどう使われるかを説明せよ。

11.35 安全なコンピュータシステムを設計するガイドラインとして通常用いられる重要な設計原則は何か。これらの設計原則が重要な理由を説明せよ。

参考文献

[Seberry and Pieprzyk 1989] Seberry, J., and pieprzyk, J., Cryptography: An Introduction to Computer Security, Prentice-Hall, Englewood Cliffs, NJ (1989).

[Shankar 1977] Shankar, K. S., "The Total Computer Security Problem: An Overview," IEEE Computer, Vol. 10, pp. 50-62, 71-73 (1977).

[Silberschatz and Galvin 1994] Silberschatz, A., and Galvin, P. B., Operating System Concepts, 4th ed., Addison-Wesley, Reading, MA (1994).

[Simmons 1992] Simmons, G. J. (Ed.), Contemporary Cryptology, IEEE, New York (1992).

[Simmons 1994] Simmons, G. J., "Cryptanalysis and protocol Failures," Communications of the ACM, Vol. 37, No. 11, pp. 56-65 (1994).

[Singhal and Shivaratri 1994] Singhal, M., and Shivaratri, N., G., Advanced Concepts in Operating Systems, McGraw-Hill, New York (1994).

[Skardhamar 1996] Skardhamar, R., Virus Detection and Elimination, Academic Press, San Diego, CA (1996).

[Smart 1994] Smart, R. K., "The X.509 Extended File System," In: Proceedings of the ISOC Symposium on Network and Distributed System Security, Internet Society, Reston, VA(February 1994).

[Stallings 1994] Stallings, W., "Kerbros Keeps the Enterprise Secure," Data Communications Magazine, pp. 103-111 (October 1994).

[Stallings 1995] Stallings, W., Network and Internetwork Security: Principles and Practice, Prentice-Hall, Englewood Cliffs, NJ (1995).

[Stalllings 1996] Stallings, W. (Ed.), Practical Cryptography for Data Internetworks, IEEE Computer Society Press, Los Alamitos, CA (1996).

[**Steiner et al. 1988**] Steiner, J. G., Neuman, B. C., and Schiller, J. I., "Kerberos: An Authentication Service for Open Network Systems," In: Proceedings of the Winter 1988 USENIX Conference, USENIX, Berkeley, CA, pp. 191-202 (February 1988).

[**Tardo and Alagappan 1991**] Tardo, J. J., and Alagappan, K., "SPX: Global Authentication Using Public Key Certificates," In: Proceedings of the IEEE Symposium on Research in Security and Privacy, IEEE Press, New York, NY, pp. 232-244 (1991).

[**Tanenbaum 1987**] Tanenbaum, A. S., Operating Systems: Design and Implementation, Prentice-Hall, Englewood Cliffs, NJ (1987).

[**Tanenbaum 1992**] Tanenbaum, A. S., Modern Operating Systems, Prentice-Hall, Englewood Cliffs, NJ (1992).

[**Tanenbaum et al. 1986**] Tanenbaum, A. S., Mullender, S. J., and van Renesse, R., "Using Sparse Capabilities in a Distributed Operating System," In: Proceedings of the 6th International Conference on Distributed Computing Systems, IEEE Press, New York, NY, pp. 558-563 (May 1986).

[**Tsai et al. 1990**] Tsai, C., Gligor, V. D., and Chandersekaran, C. S., "On the Identification of Covert Storage Channels in Secure Systems," IEEE Transactions on Software Engineering, Vol. SE-16, No. 6,pp. 569-580 (1990).

[**White et al. 1996**] White, G. B., Fisch, E. A., and Pooch, U. W., Computer System and Network Security, CRC, Boca Raton, FL (1996).

[**Wobber et al. 1994**] Wobber, E., Abadi, M., Burrows, M., and Lampson, B., "Authentication in the Taos Operating System," ACM Transactions on Computer Systems, Vol. 12, pp. 3-32 (1994).

[**Woo and Lam 1992**] Woo. T. Y. C., and Lam, S. S., "Authentication for Distributed Systems," IEEE Computer, Vol. 25, No. 1, pp. 39-52 (1992).

[**Worm 1989**] "Special Section on the Internet Worm," Communications of the ACM, Vol. 32, No. 6, pp. 677-710 (1989).

インターネット参考文献

コンピュータセキュリティに関する参考文献は、以下のところで見つけることができる。

ftp:ftp.cs.umanitoba.ca/pub/bibliographies//Misc/security.html
ftp:ftp.cs.umanitoba.ca/pub/bibliographies//Misc/security.1.html
ftp:ftp.cs.umanitoba.ca/pub/bibliographies//Misc/security.2.html
http:fulmara.ce.chalmers.se/Security/sec_bib.html
http:www.telstra.com.au/pub/docs/security

コンピュータネットワークの刊行リストは、以下のところで見つけることができる。
http://www.crpht.lu/CNS/html/PubServ/Security/bibliography.html

暗号技術に関する参考文献は、以下のところで見つけることができる。
http://mnementh.cs.adfa.oz.au/htbin/bib_lpb
(これは、暗号技術やコンピュータセキュリティの様々な側面の参考文献を有する Lawries Cryptography Bibliography へのインデックスにたどり着くためのゲートウェイである。)

暗号技術、インターネットセキュリティ及びケルベロスに関する参考文献は、以下のところで見つけることができる。
http://www.comp.vuw.ac nz/ sai/docs

COAST(Computer Operations, Audit, and Security Technology) は、Purdue 大学の計算機科学部における、コンピュータセキュリティ関連のプロジェクト群の総称である。各種 COAST プロジェクトの刊行リストと COAST の研究者によるその他のセキュリティ関連の論文は以下のところでみつけることができる。

http://www.cs.purdue.edu/coast/coast-library.html

第12章

事例研究

12.1 はじめに

本章では、実際の分散オペレーティングシステムに関して述べる。これによって、これまでの章で紹介した様々な概念について読者はより一層の理解を深めることができる。本章で説明するシステムは、Amoeba、V-System、MachおよびChorusである。最初の2つは大学における研究プロジェクトである。あとの2つもまた研究プロジェクトとして始まったが、現在では製品化されている。これらのシステムは現在も様々な開発と改良が行われている。したがって、各システムに関する完全で詳細な説明はここでは行わない。設計の目的、システムアーキテクチャ、そして重要で価値のある特徴についてのみ焦点を当てて説明する。また、説明の順序は厳密には年代に則したものではなく、システムの重要性を反映したものでもない。参考文献には、他の幾つかのシステムに関する参照を提供している。それらのシステムは分散オペレーティングシステムに関する研究活動を行っている人にとって興味深いものであろう。

12.2 AMOEBA

Amoebaはマイクロカーネル準拠の分散オペレーティングシステムであり、オランダのアムステルダムにあるVrije大学と数学コンピュータサイエンスセンターにおいて開発された。Amoebaは1981年Andrew S. Tanenbaumによって分散並列コンピューティングの研究プロジェクトとして始まった。それから、幾年かを経て魅力的な特徴を持つ分散オペレーティングシステムへと発

展した。以下の内容は [Tanenbaum 1995、Coulouris et al. 1994、 Mullender et al. 1990、Mullender and Tanenbaum 1986] をもとに記述している。

12.2.1　設計の目的と主な特徴

Amoeba の設計は以下に述べる研究と設計目標に影響を受けている。

透過性

ユーザに単一システムのイメージを提供することは Amoeba の主な目標の 1 つであった。この目標を達成するための最も重要な設計方針は、プロセッサプールモデルを使うことであった。プロセッサプールモデルでは、“ホームマシン” の概念が存在しない。すべての資源は全体としてシステムに属することになる。

並列プログラミングのサポート

透過性は分散システムの多くのユーザにとって便利な特徴であるが、分散並列アルゴリズム、言語、ツール、アプリケーションを利用するためのテストベッド (testbed) としてシステムを利用することに興味を持つユーザもいる。Amoeba はそのようなユーザのために基礎にある並列性を利用できるようにしている。このために、Orca 言語 [Bal 他 1992] が設計され Amoeba 上に実装された。

ケーパビリティ準拠、オブジェクト指向アプローチ

Amoeba のもう 1 つの主な設計目標は、分散システムを構築するのにケーパビリティ準拠で、オブジェクト指向のアプローチを使うことの可能性を調査することであった。この目標を達成するために、オブジェクトとケーパビリティが Amoeba を設計する際の統一的な方法で使われている。特に、Amoeba のソフトウェアはオブジェクトを基本としており、オブジェクトは名前づけされており、ケーパビリティを使うことで保護されている。

スモールカーネルのアプローチ

Amoeba の設計におけるもう 1 つの目標は、カーネルのサイズを最小化し、柔軟性を高めることである。この目標を達成するために、設計はマイクロカーネルのアプローチに基づいている。つまり、Amoeba において、幾つかの標準的なサービスは、カーネルの外であるユーザ空間において構築されている。こ

れによって柔軟性を高めることができる。なぜなら、これらのサービスは簡単に修正でき、異なるユーザのニーズを満たすために、同じシステム上で複数の版のサービスを同時に実行することが可能であるからである。

高性能

高性能もまたAmoebaにおける目標であった。この目標に影響された設計に関する3つの方針は、プロセッサプールモデルの利用、マルチスレッド化されたプロセッサの利用、ブリットファイルサービスの利用である。ブリットファイルサービスはディスクとキャッシュの両方に連続したバイト列として不揮発性のファイルを格納する。

高信頼性

Amoebaは高い次元の信頼性を提供するようにも設計されている。次の設計方針によって、システム全般の信頼性が改善されている。

1. プロセッサプールモデルの利用。プロセッサプールモデルでは、プロセッサを動的にプールに加えることができたり、またプールから削除することができる。したがって、幾つかのプロセッサがクラッシュした場合、幾つかのジョブはリスタートしなければならないし、システムの計算能力は一時的に低下する。しかし、そうでない場合は、システムは正常に機能し続け、高い次元のフォールトトレラント性を提供することができる。

2. 信頼性のあるプロセス間通信機構のサポート。RPCはAmoebaにおいて基本的なIPC機構である。AmoebaのRPCは最大一回セマンティクスをサポートしている。したがって、サーバがクラッシュしすばやくリブートした時でさえRPCは1回より多く実行されることはない。

3. ディレクトリサービスの信頼化。ディレクトリサービスの主な機能は人間指向のオブジェクト名からシステムが定義したケーパビリティへのマッピングを提供することである。ディレクトリサービスはAmoebaにおいて必須の構成要素である。なぜなら、ほとんどすべてのアプリケーションはそれが必要とするケーパビリティを見つけるためにディレクトリサービスに依存しているからである。もし、ディレクトリサービスが停止した場合、すべてが停止してしまうであろう。単一サイト障害によってダウンすることがないように、ディレク

トリサービスは安定記憶装置という技法を使うことでフォールトトレラントに実装された。

UNIX エミュレーション

Amoeba は既存のシステム (例えば UNIX) からではなくゼロから開発された。この設計のアプローチに潜む動機は、既存のシステムとの互換性について心配することなく、新しいアイディアを確かめることであった。しかし、この設計の結果、Amoeba のインタフェースは UNIX のそれとは全く異なるものとなってしまった。したがって、Amoeba のために何百ものユーティリティやアプリケーションプログラムをゼロから書くことを避けるために、UNIX エミュレーションのパッケージが後になって Amoeba に加えられた。このパッケージは、ライブラリの形式で提供されており、これによりほとんどの UNIX プログラムは、少しあるいは全く修正を必要とすることなく、Amoeba の上で動作することができる。Amoeba の UNIX エミュレーションに関する詳細は [Mullender et al 1990] を参照せよ。

12.2.2　システムアーキテクチャ

図 12.1 に示すように、Amoeba のハードウェアは次の主な要素から構成される。

1. **プロセッサプール** (processor pool):Amoeba はプロセッサプールモデルに基づいている。Amoeba システムは 1 つ又は複数のプロセッサプールを持つことができる。プールにある CPU は全体としてシステムに属しており、特定の "所有者" を持たない。従って、ユーザは特定のマシンにログオンするのではなく、全体としてシステムにログオンする。ユーザがアプリケーションを持っており、それを動作させる場合、オペレーティングシステムはそのアプリケーションに対してプールから 1 つ又は複数の CPU を動的に割り当てる。ユーザアプリケーションの実行が完了すると、割り当てられていた CPU は他の仕事のためにプールに戻される。ユーザがアプリケーションを実行しようとした時、もし、どの CPU も空いていない場合、CPU は時分割され、新しく到着したジョブは最も軽い負荷の CPU に割り当てられる。Amoeba は異機種性をサポートするように設計された。従って、プールにある CPU は異なるアーキテクチャのマシンでも良い。

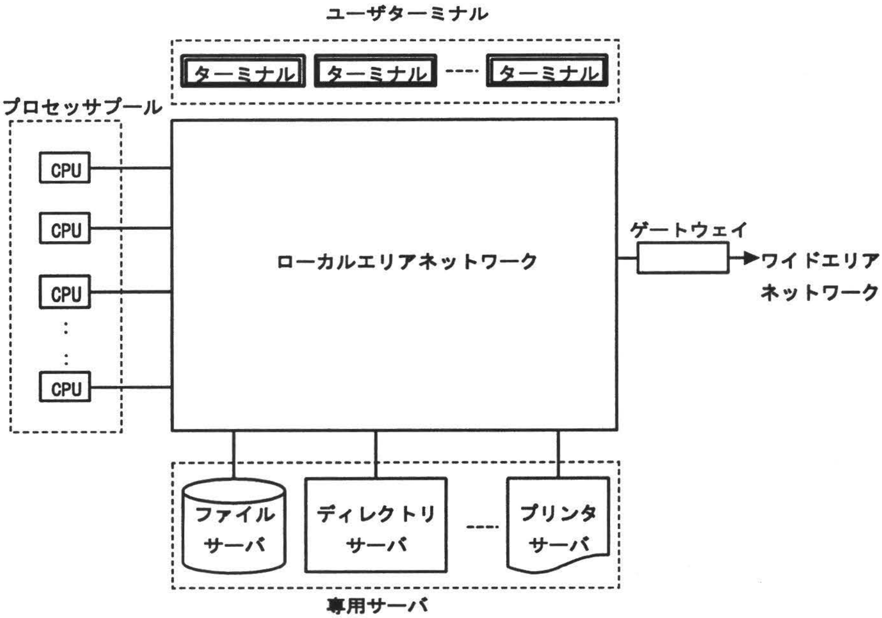


図 **12.1** Amoeba システムのハードウェアアーキテクチャ

2. ターミナル (terminal):ターミナルはシステム資源にユーザがアクセスするためのインタフェースを提供する。ターミナルは通常の X ターミナルかパーソナルコンピュータ、あるいは X ウィンドウが動作しているワークステーションである。パーソナルコンピュータやワークステーションがターミナルとして使われている場合、(コマンドインタプリタやエディタのような) ユーザインタラクションが中心の処理を要するプロセスはターミナルで実行される。しかし、ほとんどのアプリケーションはユーザとそれほど多くのインタラクションを行うことはなく、プロセッサプールの 1 つ又は複数の CPU によって実行される。

3. 専用サーバ (specialized server):専用サーバとは通常でない資源要求を持つ専用プロセスを実行するマシンである。例えば、1 つ又は複数のディスクを有するマシン上でファイルサーバプロセスを実行するのは自然であるし、1 つ又は複数のプリンタを有するマシン上でプリントサーバを実行するのも自然である。主なサーバは、ディレクトリサーバ、ファイルサーバ、ブロックサーバ、

データベースサーバ、ブートサーバである。

4. **ゲートウェイ (gateway)**:ゲートウェイは広域ネットワークにまたがる 2 つ又はそれ以上の Amoeba システムを 1 つの統一的なシステムとして接続するために利用される。

ソフトウェアアーキテクチャ
Amoeba ソフトウェアアーキテクチャは次の主な要素から構成される。

1. **マイクロカーネル (microkernel)**:Amoeba システムのすべてのマシンで動作するマイクロカーネルは、低レベルのメモリ管理サポートを提供し、マルチスレッドで動作するプロセスを管理し、プロセス間通信をサポートし、低レベルの I/O を処理する。

2. **サーバー群 (collection of servers)**:すべての他のサービス (マイクロカーネルに含まれない機能) はサーバと呼ばれるユーザレベルのプロセスによって提供される。サーバは従来のオペレーティングシステムのほとんどの機能を提供する。典型的な場合、サーバはシステムプログラマによって書かれているが、望むならばユーザは自由に自分自身のサーバを記述できる。Amoeba における幾つかの標準的なサーバは、ファイルの管理を行うビュレットサーバ、ファイルや他のオブジェクトの名前付けを管理するディレクトリサーバ、オブジェクトの自動的な複製を管理する複製サーバ、どのプロセスがどのプロセッサで動作するかを決定する実行サーバである。すべての標準サーバに対してライブラリ中にスタブ手続きが存在する。サーバを使うためには、通常、クライアントはそのスタブを呼び出すだけである。スタブはパラメータを結集し、メッセージを送信し、応答が返ってくるまでブロックする。この機能によって、サーバの実装の詳細がクライアントには見えないようになっている。

12.2.3　オブジェクトとその管理

Amoeba はオブジェクト指向のシステムであり、ソフトウェア全体はオブジェクトとして構造化されている。オブジェクトはカプセル化されたデータとそれに対して定義された操作から構成される抽象データ型のようなものである。Amoeba のオブジェクトは本来は受け身である。つまり、オブジェクトは

ビット数	48	24	8	48
フィールド	サーバポート	オブジェクト番号	権利	検査

図 **12.2** Amoeba のケーパビリティ

自ら何かを実行することはない。従って、各オブジェクトはサーバプロセスによって管理されており、サーバプロセスがそのオブジェクト上の操作を実行する。このような方法でサポートされているオブジェクトの典型的な例は、ファイル、ディレクトリ、メモリセグメント、スクリーンウィンドウ、プロセス、ディスク、テープドライブである。

オブジェクトの名前付けと保護

Amoeba の各オブジェクトはケーパビリティによって識別され保護される。図 12.2 に示すように、ケーパビリティは次の欄から構成される。

1. **サーバポート** (server port):これは 48 ビットの論理的なアドレスであり、ケーパビリティによって参照されるオブジェクトを管理するサーバを識別するアドレスである。Amoeba において、サーバをアクセスする唯一の方法はポートを通じてのみである。サーバポートは特定のマシンではなく特定のサーバ (またはあるサービスを提供するサーバ群) に関連付けられた論理アドレスであるということを忘れてはならない。従って、サーバが新しいマシンに再配置された場合、そのサーバポートがは引き継がれる。サーバはそれ自身のポートアドレスを選択することができる。

2. **オブジェクト番号** (object number):典型的な場合、サーバは同じタイプの幾つかのオブジェクトを管理している。例えば、ファイルサーバは通常何百ものファイルを管理している。オブジェクト番号の欄は 24 ビットの識別子であり、管理するオブジェクトの中で特定のオブジェクトを区別するために使われる。サーバポートとオブジェクト番号の欄は、両方でシステム全体でのオブジェクトを一意に識別する。

3. **権利** (right):これは 8 ビットの欄であり、このケーパビリティの所有者が、このケーパビリティによって識別される操作に対して実行できる操作の種類を

表している。この欄の意味は各オブジェクトタイプによって異なる。なぜなら、正当な操作自身がオブジェクトタイプによってかなり異なるからである。

4. **検査** (check):この欄はケーパビリティの正当性のために使われる。この欄は 48 ビットの数であり、ケーパビリティを偽造から保護するのに使用される。

ケーパビリティは完全にユーザプロセスによって管理される。また、暗号化によって保護される。特に、オブジェクトを作成するために、クライアントは適切なサーバに対して要求メッセージを送信する。そして、サーバはオブジェクトを作成し、ケーパビリティをクライアントに返す。このケーパビリティは**所有者ケーパビリティ**(owner capability) と呼ばれ、すべての権利ビットがはじめはオンになっている。新しく作成されるオブジェクトに対するケーパビリティを作成する際、サーバは 48 ビットの乱数を生成し、ケーパビリティの検査欄と自分自身の内部的なテーブルにその値を格納する。クライアントはこのケーパビリティをオブジェクトに対する操作の要求と一緒に送らなければならない。要求された操作を実行する前に、サーバは内部テーブルに格納された検査欄の値と送られてきたケーパビリティの検査欄の値を比べて、提示されたケーパビリティの正当性を検証する。

オブジェクトの所有者が他のユーザに対してオブジェクトに対するアクセス権を制限したい場合、オブジェクトの所有者はサーバに対して制限された権利を持つケーパビリティを作成するようにメッセージを送る。このメッセージには、オブジェクトに対する所有者ケーパビリティと新しく作成する権利のビットマスクなどが含まれる。サーバは内部テーブルにある検査欄の値と新しい権利の値の XOR をとり、結果の値 (x) を一方向関数 $[y = f(x)]$ で暗号化する。この関数は x を与えられた場合、y を求めるのは簡単であるが、y が与えられた場合、膨大な検索を行わない限り x は求まらないという特性を持つ。次にサーバは新しいケーパビリティを作成する。このケーパビリティのサーバポートとオブジェクト番号は、所有者ケーパビリティと同じ値である。また、権利操作欄は新しいビット値であり、検査欄は先ほどの一方向関数の出力の値である。この新しいケーパビリティは呼び出し側に返される。そして、呼び出し側はもう 1 つのプロセスに対して制限された権利を渡すためにそのケーパビリティを送る。

Amoeba のケーパビリティ準拠の名前付けの枠組みは十分に位置透過であるということを注意しておく。なぜなら、オブジェクトに対する操作を実行する

ためには、オブジェクトの場所やそのオブジェクトを管理しているサーバの場所を知る必要がないからである。サーバと会話するためのプロトコルは、ローカルであっても、遠隔であっても、すべての場合において同一である。従って、クライアントは何をしたいのかということに専念でき、オブジェクトがどこにあるとか、サーバがどこで実行されているとかに関しては気にする必要がない。

Amoebaのケーパビリティ準拠の保護の枠組みは、ほとんど管理的なオーバヘッドが必要ないという利点を持っている。しかし、Amoebaにおいてオブジェクトを作成し、そのケーパビリティを失うことが可能であるということに注意されたい。従って、もやはアクセスできる必要のないオブジェクトを特定し破棄するための何らかの機構が必要である。このため、Amoebaの各サーバは定期的にガーベジコレクタを実行する。ガーベジコレクタはnガーベジコレクションサイクルで使われていないすべてのオブジェクトを削除する。更に、安全でない環境では、ケーパビリティをネットワーク上の侵入者に対してあらわにすることを防ぐために、更に暗号化の手段(例えば、リンク暗号化)が必要になるであろう [Tanenbaum 1995]。

サーバ位置特定の機構

オブジェクトに対する操作を実行するために、クライアントプロセスはシステムに対してそのオブジェクトに対するケーパビリティを提示する。システムはオブジェクトに対する操作を実行するために、この要求を適切なサーバに転送する。しかし、ケーパビリティのサーバポート欄はオブジェクトを管理しているサーバを指定しているが、関連するサーバの所在に関する情報は含まれないことに注意してほしい。したがって、適切なサーバの場所を見つけるために位置特定機構が必要である。

Amoebaにおけるサーバを位置特定する機構は同報問い合わせに基づいている。すなわち、サーバの位置を特定するのに、「あなたはどこ？」というメッセージがネットワーク上に同報される。このメッセージにはサーバのポートアドレスが入っている。同報メッセージの応答がそのポートアドレスを持つサーバから返された後に、サーバの住置がわかる。この応答メッセージにはサーバのネットワークアドレスが含まれる。同報を行っているカーネルは将来のために(ポート、ネットワークアドレス)の組をキャッシュする。したがって、サーバのポートがキャッシュにないときだけ、同報が実行されることになる。

名前決定機構

ケーパビリティは低レベルのシステム指向のオブジェクト名である。これは人間が記憶し使用するのは難しい。したがって、Amoeba のオブジェクトは人間指向の ASCII 名を持っている。人間指向のオブジェクト名をシステム指向の名前 (ケーパビリティ) にマッピングするプロセスは名前決定として知られている。Amoeba では、**ディレクトリサーバ** (directory server) が名前決定の操作を実行するために使われている。ディレクトリサーバはこのためディレクトリオブジェクトを管理している。最も単純な形式では、ディレクトリは (ASCII 名、ケーパビリティ) の組の集合である。ディレクトリオブジェクトに対しては lookup、enter、delete という 3 つの基本的な操作が許されている。最初の操作はディレクトリ内のオブジェクトの ASCII 名を検索し、そのケーパビリティを返す。他の 2 つの操作はディレクトリにエントリを登録したり削除するためのものである。ディレクトリそのものがオブジェクトであることから、ディレクトリは他のディレクトリを表すケーパビリティを持つことができる。このように、ユーザは階層的なディレクトリツリーを構築することができ、さらにはグラフ構造を構築することができる [Mullender et al. 1990]。

Amoeba のディレクトリ準拠のオブジェクト名前付けの枠組みは非常に柔軟であり、様々な方法でオブジェクトの共有を実装するのに使うことができる。例えば、次の例を考えてみる。

1. 共有オブジェクトを使う異なるユーザはそのオブジェクトに対して自分自身に合った ASCII 名を使うことができる。このため、各ユーザは彼又は彼女自身の (ASCII 名、ケーパビリティ) の組をディレクトリに登録することができる。異なるユーザの (ASCII 名、ケーパビリティ) の組は必ずしも同じディレクトリに登録する必要がないことに注意してほしい。それらは 2 つ又はそれ以上のディレクトリに置くことができ、そのオブジェクトに複数のリンクを作成することができる。更に、異なるユーザに対するケーパビリティは異なる権利を持つことができる。このため、ユーザごとに共有オブジェクトに対する異なる許可を与えることができる。

2. ユーザグループで共有されるオブジェクトは、グループメンバに対してのみケーパビリティが与えられているディレクトリに格納することができる。これは、そのようなディレクトリを作成し、共有オブジェクトに対するアクセスをそれらのメンバに制限することで可能である。このように、ディレクトリのケーパビリティは多くの他のケーパビリティに対するケーパビリティとして機

能することが可能である [Mullender et al. 1990]。

3. (ASCII 名、ケーパビリティ) という 2 つのカラムからなるディレクトリを実装する代わりに、0 カラム目に ASCII 名を持ち、1 カラム目から n カラム目までケーパビリティを持つ $(n+1)$ カラムからなるテーブルとして実装することができる。この一般化されたモデルでは、オブジェクトの n 個のケーパビリティのそれぞれが異なる保護のドメインを形成する。オブジェクトに対する異なるアクセスを持つことができる。例えば、1 つのカラムは所有者ケーパビリティを持つことができるし、1 つのカラムは所有者のグループのケーパビリティを持つことができ、1 つはパブリックなケーパビリティを持つことができる。これによって、UNIX の保護の枠組みを模擬することができる [Mullender et al. 1990]。

Amoeba の設計者はディレクトリサービスはシステム必須の構成要素であると考えていた。なぜなら、ほとんどすべてのアプリケーションはそれが必要とするケーパビリティを見つけるためにディレクトリサービスに依存するからである。もし、ディレクトリサービスに障害がおきると、システム全体が機能を停止してしまう。したがって、単一サイト障害によってディレクトリサービスが障害にならならいように、それはフォールトトレラントな方法で実装されている。特に、ディレクトリサーバは組で動作しており、それぞれは (異なるディスク上にある) ケーパビリティの自分自身の配列を持っている。これによりディレクトリサーバの 1 つが持っているディスクが損傷を受けたとき、情報が失われることを防ぐことが可能である。2 つのサーバは通信を行うことで同期をはかる。

12.2.4 プロセス管理

プロセスモデル

Amoeba のプロセスはセグメント化された仮想アドレス空間と 1 つ又は複数のスレッドから構成される。したがって、Amoeba におけるプロセスの抽象概念は、3 つの基本的なオブジェクト - プロセス、セグメント、スレッドによって実現されている。次にこれらの概要について述べる。

1. プロセス (process):プロセスは状態によって定義される。各プロセスはそ

の時点での状態を定義する**プロセス記述子** (process descriptor) を持っている。プロセス記述子の 4 つの主なパートは、ホスト記述子、ケーパビリティ、セグメント記述子、そして、スレッド記述子である。

ホスト記述子はプロセスを実行するのに適したプロセッサタイプとプロセスのメモリ要求量について記述している。プロセッサタイプは、特定の CPU アーキテクチャ、CPU のグループ、あらかじめ定義された CPU のクラスとして指定することができる。プロセスはホスト記述子で指定されているプロセッサタイプであり、かつ十分なメモリを持つプロセッサでのみ実行することができる。

ケーパビリティのパートは主にプロセスのケーパビリティとハンドラケーパビリティから構成される。プロセスを操作している各クライアントはプロセスケーパビリティを所有しなければならない。一方、ハンドラケーパビリティはプロセスの終了ステータスをその所有者に通信させるために必要である。すなわち、プロセスが終了するか、どうにもならない状態になった場合 (プロセスの状態については後述する)、所有者プロセスに対してこのイベントを RPC で通知するためにハンドラケーパビリティが使われる。

セグメント記述子のパートはプロセスのセグメントについて記述している。プロセスはアドレス空間の各セグメントに対してセグメント記述子を持っている。セグメントの記述子は仮想アドレス、長さ、アクセス制御などの情報を持っている。

スレッド記述子はプロセスのスレッドに関する記述である。プロセスの各スレッドに対するスレッド記述子が存在する。中でも、スレッド記述子はスレッドのプログラムカウンタとスタックポインタを持っている。スレッド記述子の正確な内容は CPU アーキテクチャに依存する。

2. **セグメント** (segment):セグメントは名前のついた線形なメモリセクションである。プロセスは 1 つ又は複数のセグメントを有することができる。セグメントの数はプロセスが実行され生存しつづける間、変わり続ける。

セグメントを作成するには、プロセスはカーネルに対して要求を実行する。この要求にはセグメントの初期サイズを指定する。またオプションでセグメントの初期値を与えるセグメント又はファイルを指定する。カーネルは必要な分だけメモリを割り当てて、新しく作成されたセグメントのケーパビリティを要求元へ返す。このケーパビリティは要求元によってその後のセグメントに対する操作の実行のために使われる。セグメントの初期サイズはプロセスが実行している間に変わっても良い。

プロセスがセグメントを作成、読み取り、書き込み、マップ、アンマップ、廃棄するためのシステム呼び出しが提供されている。このうち、マップとアンマップ操作は様々な異なる目的のために使うことができる。例えば、プロセスがそれらをマッピングすることで新しいセグメントをアドレス空間に付け加えたり、アンマップすることでそれらをアドレス空間から削除するために使うことができる。アンマップされたセグメントはメモリ中に残り (しかし、どのプロセスのアドレス空間の一部でもない)、ファイルと同様に読み取り／書き込みができたり、また、前回と同じプロセスのアドレス空間や異なるプロセスのアドレス空間に再マップすることができる。このため、unmap 操作はセグメントのケーパビリティを戻り値とする。この特徴によって、map と unmap 操作はプロセス間通信のためにも使うことができるようになっている。送信側のプロセスはアドレス空間のセグメントを unmap して、unmap 操作によって戻されたケーパビリティを受信側に渡す。そして、受信側ではこのセグメントを自分のアドレス空間にマップする。最後に、この 2 つの操作は 2 つ又は複数のプロセスが同時にセグメントを自分のアドレス空間にマッピングすることで共有メモリを操作するために使うことができる。

3. スレッド (**thread**):プロセスは 1 つ又は複数のスレッドを持つことができる。これらのスレッドはすべて同じアドレス空間を共有する。それぞれのスレッドは自分自身のスタックとスタックポインタ、CPU レジスタのコピー、一連の大域変数を持っている。スレッドの**大域変数** (global variable) とは、スレッドが実行するすべての手続きに対しては大域であるが、他のスレッドにはアクセスできない変数である。Amoeba はスレッドが大域変数を作成し，そして使うためのライブラリを提供している。

プロセスは実行中に新しいスレッドを作成し、そして既存のスレッドを終了させることができる。したがって、プロセス中のスレッド数はライフタイムの間変化する。新しいスレッドを作成するために指定するパラメータには実行する手続きの初期アドレスと初期スタックサイズを指定する。

スレッドはカーネルによって管理されている。優先度準拠のスレッドスケジューリング方式が使われている。この方式ではカーネルスレッドはユーザスレッドよりも高い優先度を持っている。

3 つの異なる機構がスレッドの同期のために用意されている。シグナル、ミューテックス変数、計数型セマフォである。シグナルは同じプロセス中の 1 つのスレッドから別のスレッドに対して非同期な割り込みを送るのに使う。

ミューテックス変数はバイナリセマフォと同様である。それは第 8 章で述べた内容と同じ目的を持っている。計数型セマフォはミューテックスで扱うことができないような更に一般的な同期のために使う。

Amoeba では、プロセスは新しい (子の) プロセスを作成する。これは CreateSegment を子プロセスの 3 つのセグメント (テキスト、データ、スタック) に対してそれぞれ呼び出し、各セグメントに対するケーパビリティを入手することで行われる。次に、各セグメントに対して write 操作を実行し、セグメントの初期データを与える (初期データに対するポインタが CreateSegment のパラメータとして与えられている場合、これは別ステップとしては実行されない)。最後に、親プロセスは子プロセスのセグメントに対するケーパビリティをパラメータとして MakeProcess を実行する。MakeProcess は子プロセスを作成し、そのケーパビリティを戻す。親プロセスは新しく作成された子プロセスとは並行に実行しつづける。親プロセスは子プロセスのケーパビリティを使って、子プロセスを一時停止した時に、再開始したり、廃棄したりすることができる。プロセスは任意の数の子供を作れることに注意する。子プロセスは更に子プロセスを作ることができる。この柔軟性によって、任意の数の大きなプロセス木構造を作成することができる。

どの時点においても、プロセスは 2 つの状態のうちのどちらか 1 つの状態を持っている。つまり、実行中であるか、休止中 (一時停止中) であるかのどちらかの状態である。実行中のプロセスはプロセスのケーパビリティを引数に指定して stun と呼ばれるライブラリ手続きを実行することで休止させることができる。プロセスはその親又はプロセスに対して休止させるケーパビリティを持つ他のプロセスによって休止することができる。このため、親はプロセスのケーパビリティをデバッガに与え、デバッガはインタラクティブなデバッグの間プロセスの状態を好きなように変更する。プロセスが休止状態になると、そのプロセスと通信を行おうとするすべてのものに対して、カーネル中の低レベルな通信プロトコルが「このプロセスは休止中である」というメッセージで答える。

新しいプロセスに対するプロセッサの選択

Amoeba のプロセッサプールモデルでは、新しいプロセスが作られた時、システムはそのプロセスを実行するために最もふさわしいプロセッサをプールから選択しなければならない。この決定を行うための実行サーバ (run server) と呼ばれる特別なサーバが存在する。各実行サーバは 1 つ又は複数のプロセッサプールを管理している。簡単のため、各実行サーバは 1 つのみのプロセッサ

プールを管理しているとしよう。

シェルがプログラムをするとき、それは実行サーバに対して RPC を実行する。すなわち、プログラムのプロセス記述子を送りプールからプロセッサを取り出すように依頼する。実行サーバはプロセッサを選択しプログラムを実行するために次の方法を使う。

1. プロセス記述子はプロセスが実行可能な CPU アーキテクチャに関する情報を含んでいる。したがって、実行サーバはプロセス記述子の CPU アーキテクチャとそれが管理するプロセッサプールの CPU アーキテクチャの交わりをとる。これを行うために、プロセッサプールは pooldir と呼ばれるディレクトリによって表現される。pooldir はプールにある各 CPU アーキテクチャのサブディレクトリを含んでいる。このサブディレクトリにはプールの各 CPU のプロセスサーバにアクセスするためのケーパビリティがある。交わりによって得られたプロセッサはそのプロセスを実行可能な候補として選ばれる。

2. 次に、実行サーバは選ばれたプロセッサのうちどれがそのプログラムを実行するのに十分なメモリを持っているかを検査する。このため、実行サーバは各プロセッサごとのスピード、現在の負荷、現在の空きメモリ容量を持つテーブルを管理している。このテーブルの情報はプール内の各プロセッサに対して getload と呼ばれるライブラリ手続きを実行することで連続的に更新される。プログラムを実行するのに十分なメモリを持っていないプロセッサは候補から削除される。

3. 最後に、残ったプロセッサのそれぞれに対して、実行サーバはプロセッサがその新しいプログラムに割くことのできる計算能力を見積もる。このため、分かっている全体の計算能力とそこで現在動作しているアクティブなスレッドの数を入力として発見的な方法を使う。最後に、最大の計算能力を提供できるプロセッサがそのプログラムを実行する最も適した候補として選択される。

実行サーバは選択されたプロセッサのプロセスサーバのケーパビリティを呼び出し側へ返す。そして、呼び出し側はこのケーパビリティを使って選択されたプロセッサ上にプロセスを作成する。

プロセス移動の機構

Amoeba は現在プロセス移動をサポートしていないが、設計者はそれについ

ても検討を行いて、[Mullender et al. 1990] の中でプロセス移動について述べている。それによると、プロセスを古いノードから新しいノードへ移動するプロセス移動の機構は将来次の方法によって簡単に実現することができる。

1. 最初に、移動するプロセスが動作している古いノードのプロセスサーバはプロセスを休止する。
2. そして、プロセスのプロセス記述子を RunProcess 要求によって新しいノードのプロセスサーバに渡す。
3. 新しいノードのプロセスサーバは一連の ReadSegment 要求を実行することで古いノードからプロセスのアドレス空間を取り出す。
4. この後、CreateProcess 要求によって新しいノード上でプロセスをスタートする。これによって、プロセスは新しいノードで実行を開始する。
5. 新しいノードのプロセスサーバは古いノードのプロセスサーバによって実行された RunProcess 要求に対して成功応答を戻す。この応答を受け取った古いノードのプロセスサーバは DeleteProcess 要求によりカーネルから古いプロセスを削除する。これによって、プロセスの移動は完了する。

プロセス移動が行われている最中は、プロセスは古いノードで凍っている (すなわちどうにもならなくなっている) ことに注意してほしい。したがって、そのプロセスと通信を行っているプロセスはプロセスが殺されるまで「このプロセスは休止中である」という応答を受け取る。そして、削除されたプロセスとの通信に対しては「プロセスはここではない」という応答を受け取るであろう。この応答の結果として、送信側はプロセスを見つけるために locate 操作を開始する。新しいノードにプロセスが見つかると、通常の形式でそのプロセスと通信を再開することができる。

12.2.5　ファイル管理

Amoeba は同時に複数のファイルシステムをサポートするように設計されている。標準ファイルシステムをが組み込まれている、それを好まないユーザは自由に自分自身のファイルシステムを記述し使うことができる。したがって、同じ Amoeba システムにおいて、異なるユーザのファイルは異なる、互換性のないファイルシステムによって管理することができる。この柔軟性を提供できる主な理由は、Amoeba におけるファイルシステムがサーバプロセスの集合と

して動作しているからである。標準ファイルシステムに関する説明は次のとおりである。

Amoebaの標準ファイルシステムは次のタイプのサーバから構成される。

1. ブリットサーバ (bullet server):ファイルの記憶装置と操作を行う。

2. ディレクトリサーバ (directory server):オブジェクト (ファイルを含む) の名前付けとディレクトリの管理を行う。

3. 複製サーバ (replication server):ディレクトリサーバによって管理されているオブジェクト (ファイルを含む) の自動的な複製を行う。

ディレクトリサーバに関する説明は12.2.3において既に行った。したがって、ここではブリットサーバと複製サーバについてのみ説明を行う。

ブリットサーバ

ブリットサーバは非常に高速であるように設計されているため、そう呼ばれている。実際、高性能で単純な設計がAmoebaの標準ファイルシステムの設計の際2つの主な目的であった。これらの目標を達成するために次の設計決定がなされた。

1. 不変ファイルの使用 (Use of immutable files):ファイルは一度作成され削除されない限り修正することができない。したがって、ブリットサーバはread、create、deleteの3つの基本的なファイル操作のみをサポートしている。既存のファイルを修正するためには、クライアントはケーパビリティを指定してread要求をブリットサーバへ送る。この要求の応答として、サーバはクライアントにファイル全体を送る。このファイルはクライアントのメモリにロードされる。クライアントはファイルのローカルコピーに対してすべての修正を行う。そして、ブリットサーバに対してcreate要求を実行する。この要求にはパラメータとして修正されたファイルのあるメモリへのポインタが含まれる。サーバは新しいファイルを作成して新しいケーパビリティをクライアントに戻す。元のファイルはバックアップコピーとして残すか、クライアントはdelete要求をブリットサーバに送って削除することができる。

上記の説明から、不変ファイルに対するファイルの修正には典型的な場合3つのRPC(read、create、delete)がクライアントから実行されることに注意してほしい。これによってファイルの修正は複雑で遅いものになっている。この問題を解決するために、ブリットサーバは2種類のファイルをサポート

している。1 つはアンコミット、もう 1 つはコミット。**アンコミットファイル** (uncommitted file) は作成されている途中であり、(バイトの変更、挿入、削除など) 修正の要求をブリットサーバへ送ることで直接修正できるファイルである。すべての変更が完了した時、ファイルはコミットすることができる。この時点でファイルは不変なものになる。アンコミットファイルは本来遷移的なものである。したがって、読み取ることはできないことに注意してほしい。一方、**コミットファイル** (committed file) は本来永続的なものである。読み取りは可能であるが、それは不変なものであるから修正することはできない。create 要求はファイルを即時にコミットするかどうかを指定する必要がある。どちらの場合も、ファイルコピーがサーバで作成され、ファイルに対するケーパビリティがクライアントに戻される。

2. **連続バイト列としてファイル格納** (Storing files as continuous byte strings): 不変ファイルの最終的な大きさは作成時には常にわかるようになっている。なぜなら、作成した後はファイルは修正することができないからである。この属性のため、Amoeba におけるファイルはディスク及び主記憶キャッシュ上で隣接して格納される。この戦略は外部的なフラグメンテーションによって領域を無駄に使ってしまうが、メモリ及びディスクの両方において、高性能及び単純な設計という目標を達成する助けとなっている。なぜなら、隣接して格納されたファイルは 1 つのディスク操作でメモリに読む出すことができ、1 つの RPC 応答メッセージによってユーザに送ることができる。更に、この戦略によって、ファイルは初期アドレスと長さによって単純に表現することができる。これによりメモリとディスクの両方において記憶装置空間の管理が簡単になる。

図 12.3 に示すように、ブリットサーバはそれが動作している間、メモリ中におさまっているファイルテーブルを管理している。ファイルテーブル全体がブリットサーバがロードされる時にメモリ中にロードされる。

ファイルテーブルはブリットサーバによって管理されている各ファイルごとに 1 つのエントリを持っている。エントリにはファイル番号欄というインデックスがある。初期アドレスとサイズでファイルを表現するのに十分であるから、ファイルテーブルのエントリにもまた次の 3 つの欄がある。

- ファイルサイズを持つ長さの欄
- ディスク上にあるファイルの初期アドレスを持つディスクアドレス欄
- メモリアドレス欄。ファイルが現在メイン メモリキャッシュにある場合、



図 12.3 ブリットサーバの実装

この欄はメインメモリキャッシュ中のファイルの開始アドレスを持つ。

これらの欄に加えて、アクセス制御と管理を目的とする情報を持つ欄がある。

ファイルを読むために、クライアントはブリットサーバにRPCを実行する。この際、ファイルに対するケーパビリティが送られる。そして、サーバはケーパビリティのオブジェクト番号の欄からファイル番号を抽出し、ファイルテーブル中の適切なエントリを見つけ出す。各ファイルテーブルエントリもまたケーパビリティの検査欄の中で使われる乱数が格納された欄を持っている。この情報はケーパビリティの正当性を検査するために使われる。もしケーパビリティが正当であることがわかった場合、サーバは最初にファイルが現在メインメモリキャッシュにあるかどうかを見るためにファイルテーブルのメモリアドレス欄にある情報を検査する。もし、メモリ中になかった場合、サーバは次にディスクアドレス欄と長さの欄にある情報を使って、ファイル全体をディスクからメインメモリキャッシュへ取り出す。最後に、ファイル全体をRPCメッセージの応答としてクライアントに戻す。**最新アクセス最小 (least recently used; LRU)** の方法がキャッシュ管理に使われている。

ファイルのケーパビリティが失われた場合、アクセスすることのできないファイルとしてシステムに永久に残る。この状況を避けるために、アンコミットファイルとコミットファイルは異なる方法で扱われる。アンコミットファイルでは、10分間それがアクセスされない場合、単純にそのファイルは削除され、対応するファイルテーブルのエントリが解放される。削除されたファイルに対するエントリが他のファイルのために再利用されてしまった後、削除されたファイルに対する要求が到着すると、ファイルが既に変わっていることが検

出され、既に削除されたファイルに対する操作は拒否される。このアプローチは、アンコミットな状態にあるファイルは通常数分しか存在しないであろうという設計者の仮定に基づいている。

コミットファイルに対しては、ガーベジコレクタが定期的に実行され、n ガーベジコレクションのサイクルで使われていなかったファイルを削除する。このアイディアは次のように実装されている。ファイルテーブルのエントリは各ファイルに対してカウンタを持っている。カウンタの値はあらかじめ決められた定数 MAX_LIFETIME に初期化される。そして、次の 2 つの呼び出しが定期的に実行され、決してアクセスすることのできないコミットファイルを検出し削除する。

1. Age。age 呼び出しは新しいガーベジコレクションのサイクルを開始し、各ファイルテーブルエントリのカウンタ欄の値を 1 だけ減数する。この結果、カウンタの値が 0 になったファイルは削除され、それが使っていたディスク空間、キャッシュ空間、そしてファイルテーブルエントリの領域が解放される。

2. Touch。touch 呼び出しは使われているファイルが削除されるのを防ぐ。age がすべてのファイルに適用されるのとは違い、touch は特定のファイルにだけ適用される。特定のファイルにこれが呼ばれると、ファイルエントリのカウンタ欄の値を MAX_LIFETIME にリセットする。touch は任意のディレクトリにあるすべてのファイルに対して定期的に呼び出される。そして、それらが age 呼び出しによって削除されることから守っている。

複製サーバ

複製サーバはディレクトリサーバによって管理されているオブジェクト (ファイルを含む) の自動的な複製を扱う。常にバックグラウンドで稼動しており、オブジェクトの複製を作成するのに**遅延複製 (lazy replication)** のアプローチをとっている。ディレクトリシステムの任意のオブジェクトに対して複製作成の必要があるかどうかを検査するために、複製サーバは定期的にディレクトリシステムの指定された部分をスキャンする。特に、n 個のケーパビリティがあるはずなのに、m 個 ($m < n$) のケーパビリティしかないディレクトリがある場合、$n - m$ の対応するオブジェクトのコピーが追加作成されなければならない。したがって、複製サーバがそのようなディレクトリエントリを見つけた場合、関係するサーバにコンタクトして、対応するオブジェクトの複製を追加作成するためにまとめる。

オブジェクトの複製作成に加えて、複製サーバはガーベジコレクションに対しても責任を持っている。すなわち、複製サーバはディレクトリ内の各オブジェクトに対して定期的に touch 呼び出しを実行する。これによって、オブジェクトのライフタイムを更新し、それらはガーベージコレクタから削除されないようにする。また、複製サーバは定期的に age メッセージを異なるオブジェクトタイプのサーバへ送信する。これによってサーバはすべてのオブジェクトのカウンタ欄を age 呼び出しによって減数し、0 に達したカウンタを持つオブジェクトを削除する。

12.2.6 プロセス間通信

基本的な IPC 機構

Amoeba はオブジェクトの操作に対してクライアントサーバモデルを使っている。すなわち、クライアントプロセスはオブジェクト上の操作を実行するためにサーバプロセス (オブジェクト管理者) に対して要求を送る。サーバはクライアントの要求メッセージを受け取り、要求のあった操作を実行し、クライアントに応答を返す。各標準サーバはクライアントが呼び出すことができるスタブ手続きの集合によって定義されている。スタブ手続きはサーバが提供するサービスとそのパラメータを正確に定義している。スタブ手続きがクライアントによって呼び出されると、パラメータがメッセージの中に詰め込まれ、カーネルのプリミティブを起動してそのメッセージを実際に送る。カーネルは次の 3 つの基本的な IPC プリミティブを提供している。

■ trans はクライアントがサーバに何らかの仕事をしてもらうために要求をサーバに送るために使う。

■ get_request はサーバが指定したポートへ届いたメッセージを受け付ける意志があることを表明するために使う。

■ send_reply はサーバがクライアントに応答を送るのに使う

これらのプリミティブは実際にはメッセージパッシングに関係しているが、スタブ手続きによって提供されている手続きのインタフェースはこの基本的な IPC 機構をプログラマに対して RPC(遠隔手続き呼び出し) として見えるようにしている。したがって、Amoeba の基本的な IPC 機構は RPC としても参照される。この RPC 機構は次のような属性を持っている。

1. 同期型の通信のみをサポートしている。つまり、trans 呼び出しを行った後、クライアントスレッドは呼び出されたサーバから対応する応答が返ってくるまでブロックする。同様に、get_request を呼び出した後は、サーバはスリープし、要求メッセージが来るのを待っている。サーバは要求メッセージが到着した時にだけアンブロックされる。

2. メッセージはバッファリングされない。したがって、受信側がそれを受け取る準備ができない状態にある場合、メッセージは単に破棄される。この場合、送信しているカーネルはタイムアウトをし、そのメッセージを再転送する。ユーザが再転送の最大時間を指定できるように柔軟性も提供している。最大時間が過ぎると、カーネルは転送をあきらめ失敗をレポートする。

3. 最大 1 回セマンティクスをサポートしている。すなわち、たとえサーバがクラッシュしたり、急にリブートしたとしても、システムは RPC が 1 回より多くは決して実行されないことを保証している。

4. 状態を持たないサーバが使われている。したがって、各 RPC は完全に自己完結的であり、サーバのメモリに先に格納されている情報には依存しない。

グループ通信

RPC 機構に加えて、Amoeba はグループ通信の機構もサポートしている。グループは何らかのサービスを提供するのに協力する複数のプロセスから構成される。Amoeba は閉鎖的なグループいう概念を用いている。つまり、サイズとグループのメンバであるプロセスに関する情報はグループの外にいるプロセスには知られない。したがって、グループによって提供されるサービスにアクセスするためには、クライアントプロセスは単にメンバの 1 つのプロセスに対して RPC を実行するだけである。そして、そのメンバがグループ内でグループ通信を行い、他のメンバと協力してクライアントの要求に対してサービスを行う。

プロセスは動的にグループに参加したり離れたりできる。また、複数のグループのメンバに同時になることができる。システムは新しいグループを作ったり、プロセスをグループに動的に参加あるいは離脱させたり、グループのすべてのメンバにメッセージを送ったり、グループからメッセージを受け取ったりするためのプリミティブを提供している。Amoeba のグループ通信の機構は次の属性を持っている。

1. メッセージの順序つきの配送を保証している。すなわち、もし 2 つのプロ

セスが同時にグループに対してメッセージを送った場合、すべてのグループのメンバはすべておなじ順序でメッセージを受け取ることがシステムによって保証されている。グループによって受信されるメッセージを適切に順序化するためにシーケンサプロセスが使われる。シーケンサプロセスは選任アルゴリズムによって選ばれる。

2. メッセージの信頼性のある配送が保証されている。すなわち、ユーザプロセスがメッセージを同報送信した場合、例えばハードウェアがパケットをなくしたとしても、そのメッセージは正しくグループのすべてのメンバに配送される。信頼性のあるメッセージ配送を保証するために使われている基本的な機構は紛失したメッセージを再送するためのタイムアウトに基づく再送である。つまり、重複したメッセージを検出するためにユニークな識別子を使っており、確認通知が受信されていないメッセージを格納するための履歴バッファを使っている。

3. 任意の k 個のプロセス (シーケンサを含む) の紛失に耐えうる。ここで、k(復元力の度合い) はユーザがグループを作成するプリミティブでパラメータとして指定した値である [Tanenbaum 1995]。k の値が大きければ大きいほど、より冗長性が要求され、グループ通信はより遅くなる。したがって、ユーザは注意深く k の 値を選ばなければならない。

Amoeba のグループ通信の詳細については、更に [Kaashoek and Tanenbaum 1991] に見ることができる。

低レベル通信プロトコル

Amoeba の RPC とグループ通信の機構は実際のメッセージ転送に FLIP(Fast Local Internet Protocol) と呼ばれるプロトコルを用いている。FLIP の詳細は第 2 章で既に説明したので、ここでは重複は避ける。

FLIP は Amoeba で高性能を達成するために使われている。しかし、Amoeba のユーザが FLIP の代わりに標準の TCP/IP を使う必要がある場合がある。例えば、TCP/IP は X 端末と通信するのに使わなければならないし、非 Amoeba のマシンとメールを送受信するのにも使わなければならない。また、インターネットを通じて他の Amoeba システムと相互通信するにも使わなければならない [Tanenbaum 1995]。これらのタイプの通信を機能させるため、Amoeba

は TCP/IP サーバを持っている。TCP/IP 準拠の通信では、クライアントは TCP/IP アドレスを与え、TCP/IP サーバに RPC を実行する。TCP/IP サーバは要求されたプロセスとコネクションを確立し、クライアントプロセスに対してケーパビリティを返す。このケーパビリティによって、クライアントはその確立したコネクションを使うことができる。1 度コネクションが確立すると、2 つのプロセス間での続く RPC は Amoeba プロセスが TCP/IP が使われていることを知る必要なく実行できる。

通信セキュリティ

本物のクライアントであることの保証。Amoeba は本物のクライアントであることを保証するために 2 つの保護レベルを備えている。サーバに対するアクセスを保護するためのポートと各オブジェクトへのアクセスを保護するためのケーパビリティである [Mullender 1990]。すなわち、サーバのポートに関する知識と正当なケーパビリティの所有は、送信側がサーバと通信をする権利を持っており、ケーパビリティによって識別されるオブジェクトに対してそのケーパビリティが操作の実行を許すだけの権利を持っているという十分な根拠として、システムは扱う。通常のユーザプロセスが使うポートは一般にはそのユーザプロセスと通信を行う権利を持っているプロセスにのみ明らかである。したがって、パブリックなサーバの場合、ケーパビリティはクライアントの真性を保証するための基本的な機構として機能する。

本物のサーバであることのを保証。パブリックなサーバのポートはすべてのユーザに知られているため、侵入者がサーバのポート上で get_request を実行することで、サーバを偽装することは簡単である。Amoeba ではこの問題を解決するために、一方向暗号化の技法を使っている。このアプローチでは、各ポートはポートの組、put-port = F(get-port) という関係を持つ get-port と put-port によって定義される。関数 F は公開されている一方向暗号化関数である。したがって、ある get-port が与えられると put-port を計算するのは簡単である。しかし、put-port から get-port を得るのは実践的にはうまくいかない。ここで、F はケーパビリティの保護に使われる一方向暗号化関数と同じである必要はない。なぜなら、2 つの概念は関係ないからである。

プロセス作成の過程として、サーバは get-port を選択し、それを秘密にしておく。対応する put-port を計算し、それを公開する。サーバはクライアントから要求を受け付ける準備ができると、get_request(get-port, ...) を呼び出す。カーネルは対応する put-port を計算し、listen しているポートのテーブル

にそれを格納する。クライアントがメッセージをサーバに送りたいと思う場合、trans(put-port, ...) を呼び出す。そして、クライアントマシンのカーネルは put-port をヘッダ欄に持つメッセージをサーバに送る。サーバ側では、カーネルがメッセージヘッダの put-port とテーブルの put-port がマッチするかどうか比較し、そして、対応するサーバにメッセージを転送する。

この方式は、本物サーバだけがクライアントの要求を受信できることを保証する。なぜなら、get-port はネットワーク上には決して流れないし、公開されている put-port から得ることもできないからである。

12.3 V-SYSTEM

V-System はマイクロカーネル準拠の分散オペレーティングシステムであり、高性能なネットワークでつながれたクラスタ化されたワークステーションのために設計されたものである。V-System はスタンフォード大学の分散システムに関する研究プロジェクトとして David Cheriton をディレクタとして始まった。基本的には Thoth と呼ばれるそれ以前のシステムで得た経験の結果である [Cheriton et al. 1979]。相対的に低コストで、高性能なワークステーションと LAN の利用可能性が増し、機能が発展していることが、この開発の元となっている。

V-System はこれまでの何年かの間、スタンフォード大学で実際に使われてきた。スタンフォードにおける V-System の環境はイーサネットで接続された強力な Sun と VAX の集まりから構成されている。V-System はスタンフォードのライセンスのもと配布されており、幾つかの他の大学、研究所、企業などで使われている。次の V-System に関する説明は [Cheriton 1984, 1987, 1988, Cheriton et al. 1990, Berglund 1986] をもとにしている。

12.3.1 設計の目標と主な特徴

V-System の設計は次の研究と設計目標に影響されている。

高性能通信

V-System の設計者は高性能なプロセス間通信が、より良い性能と、結果としてアプリケーションシステムをより単純な設計にするために必要であるという考えを持っていた。また、本当のネットワーク透過性にも高性能なプロセス

間通信が必要であると考えていた。高速な通信機構を持つシステムでは、アプリケーションシステムがより良い性能で動作することは明らかである。アプリケーションシステムがより単純な設計になるのは、性能の低下を抑えるために、通信を高度に最適化する必要がないからである。最後に、本当のネットワーク透過性が達成できるのは、高速な通信によって場所を意識しないでも資源にアクセス することができるためである。

V-System の高速なプロセス間通信は次によって達成している [Cheriton 1988]。

1. 相対的に単純で基本的なプロセス間通信のプリミティブを使う。
2. VMTP(Versatile Message Transaction Protocol) と呼ばれるトランスポートプロトコルを使用する。これは先のプリミティブをサポートするために注意深く設計されている。
3. 性能にとって重要な共通の場合を最適化する。
4. 内部的に効率的な通信を行うようにカーネルを構築する。

統一的なインタフェースとプロトコル

V-System の設計者はソフトウェアモジュールではなく、プロトコルとインタフェースがシステムを定義すると考えていた。すなわち、デバイス独立あるいはネットワーク独立な統一的なインタフェースとプロトコルが開放型システムアーキテクチャを構築するために定義できる。したがって、V-System の設計者は主にプロトコルとインタフェースを設計することに注力した。その結果が、一連のデータ転送のためのプロトコル、名前付け、I/O、遠隔実行、移動などであり、それらが標準化のための基礎を提供している。

相対的に小さいカーネル

V-System のもう 1 つの重要な設計目標は、相対的に小さいカーネルを持つことであった。このカーネルは分散システムに対するソフトウェアのバックプレーンを提供することができる。これは、良いハードウェアバックプレーンが提供するものに類似しており、システムの他の残りの部分はマシン独立でネットワーク独立な形でユーザレベルのものとして実装する。したがって、ファイルシステム、資源管理、保護など、従来のオペレーティングシステムに見られる多くの機能はカーネルの外で動作するサーバによって提供されている。

高性能

高性能であることはV-Systemにおいて重要であった。なぜなら、設計者はエレガントかどうかに関係なく、遅いシステムは誰も使おうと思わないと考えたからである。V-Systemにおいて高性能のために使われた重要な技法の幾つかは次の通りである。

1. 既に述べた方法によってプロセス間通信を高速にする。
2. マルチスレッド化されたプロセスを使うことで、アプリケーションプログラムに並行性を使うことを許している。
3. オブジェクト名の効率的な解決のために、各プログラムの名前の接頭辞のキャッシュを管理する。
4. 共有メモリの一貫性に関して問題指向のアプローチをとっている。この方法は性能を向上するために一貫性に関して潜在的な非一貫性をアプリケーション固有の形式で扱うことによって、緩和された一貫性寛大な方法を実装している。

ネットワーク透過性

他の分散オペレーティングシステムと同様に、ネットワーク透過性がV-Systemの目標であった。この目標を達成するために、V-Systemの名前付け機能は3レベルのモデルに基づいている。文字列の名前、オブジェクト識別子、そして、エンティティ識別子である。

便利なプログラミングモデルのサポート

V-System設計のもう1つの目的は、アプリケーションプログラマが一連の手続きインタフェースを通じてシステムサービスにアクセスできるように、便利なプログラミングモデルをサポートすることであった。この目標を達成するために、システムが提供する各手続きはV-Systemの実行時ライブラリの1つとして作られている。アプリケーションプロセスがシステムサービスにアクセスするために手続きを起動した場合、可能ならば、起動された手続き自身がその要求された操作を実行する。可能でない場合は、カーネルが提供するプロセス間通信を使ってV-Systemの適切なサービスモジュールと通信してその操作を実行してもらう。どちらの場合も、アプリケーションプロセスに応答を返すのは起動された手続きである。したがって、操作の実行に使われている実際の機構はアプリケーションプロセスには透過的なものである。

UNIX エミュレーション

V-System の比較的新しい目標は既存の UNIX プログラムが動作できるように、UNIX システムとのバイナリ互換性を提供することであった。このため、UNIX エミュレーションパッケージが後になって V-System に付け加えられた。詳細は [Cheriton et al. 1990] にある。

12.3.2 システムアーキテクチャ

ハードウェアアーキテクチャ

図 12.4 に示すように、V-System のハードウェアアーキテクチャはイーサネットのような通信ネットワークによって接続された一群のワークステーションから構成される。ワークステーションはおおまかには次のように分類される。

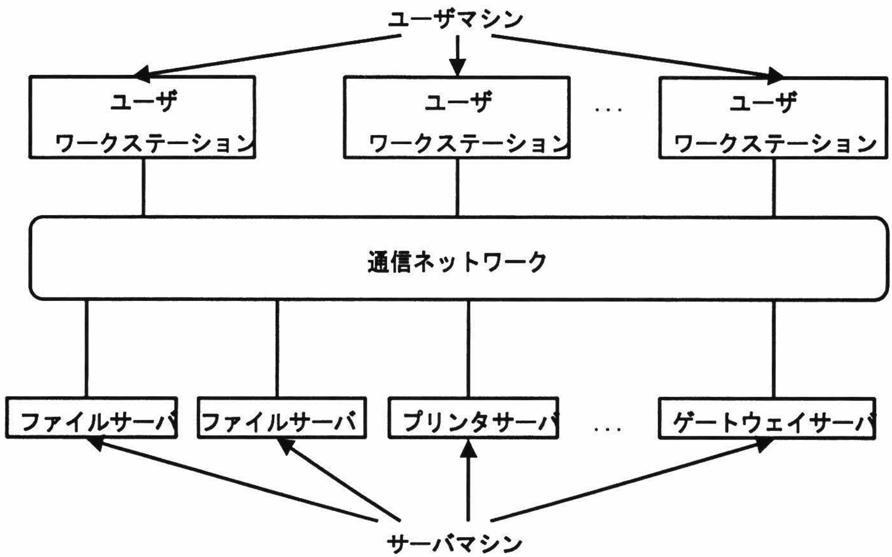


図 **12.4** V-System のハードウェアアーキテクチャ

1. **ユーザマシン** (User machines):この分類に属しているワークステーションはインタラクティブなユーザをサポートしている。ディスプレイ、キーボード、マウスに加えて、ユーザに対する資源処理の大部分を請け負う。

2. サーバマシン (Server machines):この分類に属するワークステーションは、ファイルサービス、プリントサービス、認証サービス、その他のサービスを提供する専用のサーバとして機能する。この分類に属する各ワークステーションは1つ又は複数タイプのサービスを提供することもできる。例えば、2次の記憶装置を持つワークステーションはV-Systemのファイルサーバソフトウェアを実行し、そして、独占的にファイルサービスを提供することができる。カーネルのプロセス間通信によって、このサービスと他のサービスはネットワーク上のすべてのワークステーションに対してネットワーク透過になっている。

ソフトウェアアーキテクチャ

ソフトウェアの観点からは、V-Systemのアーキテクチャは主に次の階層から構成される (図12.5)。

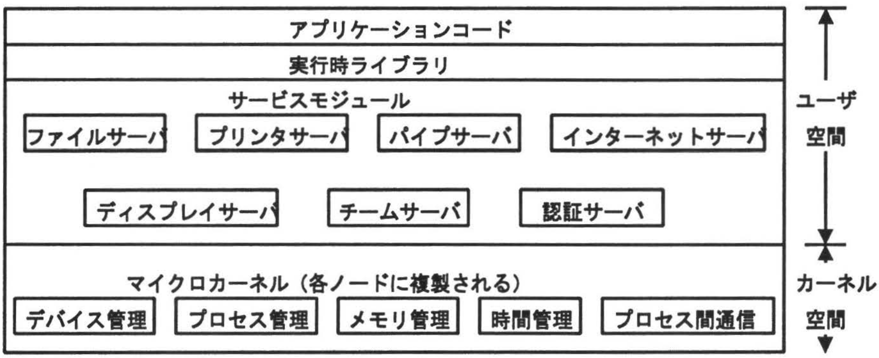


図 **12.5** V-Systemのソフトウェアアーキテクチャ

1. カーネル層 (the kernel layer):V-Systemのカーネルはソフトウェアアーキテクチャの中の最も低い層である。小さく保つために、カーネルは極小主義の哲学を用いて設計されている。すなわち、他のすべてのシステムと特徴を構築するのに十分であり強力な機能のみがカーネルの中に置かれている。したがって、カーネルは、軽量プロセスの管理、アドレス空間の管理、プロセス間通信のような機能のみを実行する。カーネルの別々のコピーがそれぞれのワークステーションで稼動している。そして、それらのすべてが協力して、アプリケーションプロセスレベルにおける単一システムイメージを提供している。

2. **サービスモジュール層** (service modules layer):この層のモジュールはユーザに対して様々なタイプのサービスを実装するために、基本的な手段でカーネルが提供するハードウェア資源にアクセスする。例えば、この層のファイルサーバモジュールはカーネルによってサポートされている直接ディスクアクセスを使って UNIX に類似したファイルシステムを実装している。他のサービスモジュールには、UNIX に類似したパイプを実装しているパイプサーバ、TCP/IP ツールを実装しているインターネットサーバ、プリントジョブをスプーリングするプリンタサーバ、ビットマップディスプレイを使ってマルチウィンド機能を実装しているディスプレイサーバがある。このように、V-System では従来はカーネルにあった多くの機能をユーザレベルのサーバプロセスとして実装することが可能である。

3. **実行時ライブラリ** (runtime libraries):この層は様式化された言語やアプリケーションからオペレーティングシステムへのインタフェースを実装している。多くの V-System アプリケーションとコマンドはこれらの様式化されたインタフェースによって記述されており、下位に存在するシステムが分散していることを意識していない。したがって、この層のソフトウェアによって、V-System はアプリケーションプログラマに対して、システムサービスへのアクセスを提供する一連の手続きインタフェースとして見える。また、非分散システムで開発された多くのプログラムを単にこの層の実行時ライブラリにリンクすることで、少し又は全く修正することなく V-System 上へ移植できる。

4. **アプリケーションコード層** (application code layer):この層にはアプリケーションプログラムのコードが存在する。

12.3.3　オブジェクト名前付け

V-System のオブジェクト名前付けは、文字列の名前、オブジェクト識別子、エンティティ識別子として構造化された 3 レベルのモデルに基づいている。名前の使用や管理や名前を与えられたオブジェクトの位置特定は以下に述べる。

文字列の名前

文字列の名前は人間中心の名前である。それは主にファイルなどの永続的なオブジェクトを名前付けするために使われる。V-System では、各オブジェク

トはオブジェクト管理者によって管理されている。オブジェクト管理者は通常同じ種類の幾つかのオブジェクトを実装し管理している。例えば、ファイル管理者は幾つかのファイルを実装し管理している。各オブジェクト管理者は自分自身が管理するオブジェクトのために1つ又は複数のディレクトリを管理している。ディレクトリエントリにはオブジェクトの文字列の名前とオブジェクトへのポインタがある。各ディレクトリは文字列の名前からなる名前空間のコンテキストを形成する。したがって、文字列の名前は常にあるコンテキストで解釈され、各オブジェクト管理者は自分自身のコンテキストを管理している。

各オブジェクト管理者はそれが管理しているコンテキストに対して大域的に一意な名前を付け、自分自身を**名前処理 (プロセス) グループ** (name-handling (process) group) に加える。コンテキスト名はそのコンテキストで定義される文字列名に対する名前の接頭辞として機能する。すべての文字列名は何らかのコンテキストに対して相対的に解釈されるので、(コンテキスト名、文字列名) の組は一意にオブジェクトを特定するために使うことのできる**限定名**を形成する。限定名の両部分とも階層的な名前空間かフラットな名前空間のどちらかを形成している。

オブジェクトにアクセスするために、クライアントプロセスはオブジェクトの限定名を指定する。そして、名前付け処理グループ内のすべてのオブジェクト管理者に対してアクセス要求がマルチキャストされる際、そこで名前もまたマルチキャストされ、適切なオブジェクト管理者が位置特定される。各オブジェクト管理者は限定名からコンテキスト名を取り出し、名前づけられたオブジェクトが自分が管理するコンテキストに属するかどうか検査する。そして、適切なオブジェクト管理者のみが名前のついたオブジェクトに対して要求があったアクセス操作を実行し、応答をクライアントへ返す。

上記に述べたマルチキャストによる問い合わせの数を減らすために、各プロセスは名前の接頭辞をキャッシュしている。この接頭辞はコンテキスト名とオブジェクト管理者のバインディングを持っている。このキャッシュは開始名のキャッシュミスを避けるために、プロセスの開始時に初期化される。更に、名前の接頭辞のキャッシュの一貫性のために、使用時更新による機構を用いている。

上記に述べた名前管理者は、一般に、名前を対応するオブジェクトの操作の一部としてマップするという利点を持っている。V-System の設計者によると、一般に、このようなオブジェクト管理と名前付けの統合には次のような利点がある [Cheriton and Mann 1989]：

1. これによって効率になる。なぜなら、オブジェクトに対する操作は、名前を指定することで要求される。名前決定の操作とそのオブジェクトに対して要求された操作は 1 セットのメッセージ交換で完了することができるからである。これは分離した名前サーバを使っているシステムとは対象的である。この場合、1 セットのメッセージ交換が名前決定を行い、オブジェクト (その管理者) の位置を特定するために必要であり、また、別のもう 1 セットのメッセージ交換が、オブジェクト上に要求された操作を実行するのに必要となる。

2. 名前決定とオブジェクトのアクセス操作の両方に一貫した信頼性を導く。すなわち、この方法では、オブジェクトの名前は常に、そのオブジェクト (その管理者) が利用可能な時に変換することができる。これは、分離した名前サーバを用いているシステムとは対照的である。そのようなシステムでは、オブジェクトが利用可能であるのに、名前サーバがダウンしていることでアクセスすることができないということになりがちである。

3. より単純な設計にすることができる。なぜなら、1 つのサーバのオブジェクトに関するすべての情報 (名前、属性、データなど) が利用可能であることで、それらの一貫性を維持することがより簡単になる。例えば、オブジェクトが削除された時、他のサーバと通信を行う必要なくすべての関連する情報を削除することができる。

4. この設計によって、オブジェクト管理者と通信し名前付けられた操作に対する情報へのアクセスを制御するセキュリティ機構を持つ自動的なアプリケーションが可能となる。

オブジェクト識別子

オブジェクト識別子はシステム中心の名前である。それは、オープンしたファイル、アドレス空間、コンテキストやディレクトリなどの遷移的なオブジェクトを識別するのに使われる。これによって、文字列名を扱うオーバヘッドやオブジェクトにアクセスするたびの検索のオーバヘッドを避けることができる。例えば、プロセスはファイルをオープンするためにファイルの文字列名を使う。この時、システムはオブジェクト識別子をオープンされたファイルに対して割り当て、プロセスにその識別子を戻す。プロセスはオブジェクト識別子を使って、その後の操作でファイルを参照する。

オブジェクト識別子は (manager-id, local-object-id) の組として構造化されている。manager-id はそのオブジェクトを実装しているオブジェクト管理者を識別するものであり、local-object-id はそのオブジェクト管理者が管理するすべてのオブジェクトの中でオブジェクトを識別するものである。manager-id の部分はオブジェクトに対する操作の要求をクライアントが効率的に正しいオブジェクト管理者に送るために使われる。一方、local-object-id の部分は、オブジェクト管理者が正しいオブジェクトに対して要求された操作を実行するために使う。

オブジェクト識別子で識別されるオブジェクト (遷移的なオブジェクト) の一生は、埋め込まれている manager-id の一生を超えてはならないことに注意してほしい。なぜなら、オブジェクト管理者の manager-id は対応するプロセスがクラッシュしたとき不正なものとなるからである (オブジェクト管理者はリブートの際新しい manager-id を割り当てられる)。

オブジェクト管理者が複数のノードで複製される、あるいは、分散している場合、グループ全体のサーバインスタンスを 1 つのグループ識別子によって識別するためにプロセスグループ機構 (後述) が使われる。この場合、グループ内の特定のサーバはグループアドレス機構 (後述) のコレジデントアドレス (限定) 機構を使ってアドレス変換することができる。

エンティティ識別子

エンティティ識別子は、プロセス、プロセスグループ、トランスポート層の通信エンドポイントを識別するために使われる。したがって、manager-id と group-id は基本的にエンティティ識別子である。

エンティティ識別子は、固定長 (64 ビット) のバイナリ値であり、この値はホストアドレスに非依存のものである (ホストアドレスはネットワーク又は相互接続されたネットワークに依存したものである)。したがって、プロセスがホストからホストへと移動する時、process-id を変更する必要はない。オブジェクト識別子からホストアドレスへのマッピングはカーネルによって行われる。このため、カーネルはマルチキャスト機構によるマッピングのようなキャッシュを使って、キャッシュに見つけることができなかったマッピングを別のカーネルに問い合わせる。エンティティ識別子がプロセスグループを識別するために使われるとき、エンティティ識別子には基本マルチキャストアドレスに分割される時そのグループのマルチキャストホストアドレスを生成する埋め込み型のサブ欄が含まれている。

プロセス管理

V-System で使われているプロセス管理のための 3 つの基本的なプロセスの抽象概念は、**プロセス** (process)、**チーム** (team)、**チーム空間** (team space) である。これらはそれぞれこの本で説明した “スレッド”、“プロセス”、“プロセスのアドレス空間” の概念に対応している。混乱を避けるために、今後の説明では、**スレッド** (thread)、**プロセス** (process)、**アドレス空間** (address space) という言葉を、プロセス、チーム、チーム空間の代わりに使う。

V-System では、プロセス管理の作業は主に 2 つのモジュールによって行われる。すなわち、**カーネルプロセスサーバ** (kernel process server) と**プログラム管理者** (program manager) によって行われる。システムの各ノードはこれら 2 つのモジュールを持っている。カーネルプロセスサーバはカーネルの中で実行する。一方、プログラム管理者はユーザ空間の中で実行する。V-System の設計者の目標の 1 つは、カーネルによるプロセス管理の作業を最小化することであり、これによってカーネルを可能な限り小さくすることができる。したがって、カーネルプロセスサーバは基本的な低レベルのプロセス管理の機能を実行するのみであり、残りはユーザレベルのプログラム管理者によって実行される。プロセス作成、破壊、スケジューリング、移動などにおけるこれら 2 つのモジュールの役割を次に説明する。

プロセス作成と終了

スレッドとプロセスは動的に作成され破壊される。これらは CreateProcess、CreateTeam、DestroyProcess のプリミティブによって行う。スレッドが作成される場合、それは作成したプロセスの一部として作成される。スレッドが破壊される場合、それによって作成されたすべてのスレッドもまた破壊される。プロセスの作成はスレッドの作成と同様である。但し、プロセスの作成では、新しく作成されたスレッドは親プロセスの一部としてではなく、別のプロセスとして作成される。また、プロセスの破壊については明示的な操作がないことに注意してほしい。プロセスではプロセス中の最後のスレッドが破壊されるとき、プロセスも自動的に破壊されるからである。

プロセス作成におけるカーネルの作業を最小化するために、V-System はプロセスの開始とアドレス空間の生成及びその初期化とを分けている。したがって、カーネルプロセスサーバの知っている限りでは、新しいプロセスの作成は単に新しいプロセス記述子を割り当て初期化することである。アドレス空間の

割り当てと初期化はV-Systemの仮想メモリシステムによって実行される。それは次に説明する。

V-Systemでは、アドレス空間は、**リージョン** (region) と呼ばれるアドレスの範囲であり、これはオープンしたファイルや統一的入出力 (UIO) オブジェクトの一部に束縛されている (UIOオブジェクトは普通のシステムではオープンしたファイルに対応している)。リージョンにの中のメモリアドレスにアクセスすることは、リージョンに束縛されているオープンしたファイル中の対応するデータをアクセスすることに対応している。**カーネルメモリサーバ** (kernel memory server) モジュールは物理的なメモリを開いたファイルのページのキャッシュとして管理している。束縛されたオブジェクトのキャッシュされていない部分に対応したリージョンの部分がアクセスされると、ページフォルトが起きる。ページフォールトを扱うために、カーネルは仮想アドレスから束縛されたUIOやオープンしたファイルのブロックへのマッピングを行う。または，フェイルをオープンし、そのブロックをカーネルのページフレームキャッシュに見つけるか、あるいは、フォールトしているプロセスにデータブロックを読むための要求を、オープンしているそのファイルを実装しているサーバへと送らせる。ファイルのような読み取り／書き込みアクセスが標準的なUIOインタフェースを使ってアドレス空間に対して実行できる。

プロセスのアドレス空間の作成と初期化は仮想メモリシステムを使うことによって単純な作業となる。まず、アドレス空間の記述子が割り当てられ、プロセスのプログラムファイルがこのアドレス空間に束縛される。このアドレス空間の一部をプロセスが参照するとき、プログラムファイルのページガ転送され要求に従って動的にアドレス空間にマッピングされる。したがって、V-Systemのカーネルはプログラムローディングのための特別な機構を持たない。

プロセスの終了におけるカーネルの仕事は次の方法で単純化されている。V-Systemでは、オープンしたファイルのような、多くのオペレーティングシステムにおける資源はユーザレベルのサーバモジュールによって管理されている。したがって、プロセスが終了した際、カーネルレベルで返還する資源はほとんどない。更に、プロセスが終了しても、カーネルは関連するサーバに通知を行わない。なぜなら、各サーバがクライアントプロセスに対して割り当てた資源を追跡し、クライアントが存在するかどうかを定期的に検査することに責任を持っており、もし、クライアントが存在していなければ、資源を返還する。例えば、ファイルサーバにはガーベージコレクタというプロセスがあり、これはもう既に存在しないプロセスに関連づけられているファイルをクローズして

くれる。

遠隔プログラム実行

V-System では、次のプリミティブの 1 つを遠隔マシンでのプログラム実行のために、コマンドインタープリタのレベルで使うことができる [Theimer et al. 1985]。

- $< program - name >< arguments > @ < machine - name >$
- $< program - name >< arguments > @*$

前者では、指定したプログラムは指定したマシンで実行される。しかし、後者では、指定したプログラムを実行するための適切なマシンを選択するのはシステムの責任になる。適切なマシンの選択は次に説明する方法によって行われる。

V-System では、すべてのノードのプログラム管理者が既知の**プログラム管理者 (プロセス) グループ** (program manager (process) group) にグループ化される。ユーザが後者のプリミティブを使って遠隔プログラムの実行を要求する場合、ユーザのマシンのプロセッサ割り当てモジュールはプログラム管理者グループに対して “遠隔プログラム実行要求” をマルチキャストする。ユーザの遠隔プログラム実行の許可と十分な量のプロセッサとメモリ資源を持ってるマシンのプログラム管理者のみがこのメッセージに対して応答する。そして、プロセッサ割り当てモジュールは一定の方針に従って、受け取った応答の中から最も適切なマシンを選択する。現在の方針では、一番最初に応答したプログラム管理者を持つマシンが選択される。なぜなら、これは一般的に最も負荷の低いマシンであり、また、この方針は単純であり、安価に実装できるからである。

プログラム管理者の選択を除けば、遠隔プログラム実行はローカルプログラム実行と同じである。なぜなら、V-System のプロセスはネットワーク透過な実行環境によって提供されているからである。更に、カーネルとプログラム管理者はこのマシンで動作しているもともとローカルなプログラムと遠隔プログラムの両方に対して同一のサービスを提供する。

プログラム実行 (ローカル又は遠隔) の開始には、(ローカルマシンまたは遠隔マシンの) 適切なプログラム管理者に要求を送ることが含まれる。これによって、新しいアドレス空間を作成し、このアドレス空間に対して指定したプログラムのイメージファイルをロードする。カーネルプロセスサーバとカーネ

ルメモリサーバの協力で、プロセス管理者はアドレス空間をセットアップし、作成者からの応答を待つ最初のプロセスを作成する。そして、プログラム管理者は新しく作成されたプロセスを要求側へ転送することで、新しく作成されたプロセスの制御を要求側へ引き渡す。要求側は、プログラムの引数、デフォルトI/O、様々な“環境変数”で新しいプログラム空間を初期化する。最後に、最初のプロセスに応答を行うことでプログラムの実行を開始する。

プログラムはもともとのマシンのハードウェアデバイスへの低レベルなアクセスを必要としない場合にのみ遠隔で実行できることに注意してほしい。すなわち、そのプログラムは、ディスク、フレームバッファ、ネットワークインタフェース、シリアルラインなどのハードウェアデバイスに、もともとのノードのデバイスサーバに直接アクセスして、アクセスすべきではない。

プロセススケジューリング

V-System では、プロセスは優先度を用いてスケジュールされる。カーネルの作業を単純化するために、2つのレベルのスケジューリングが使われており、これによってノードのプロセッサをそのノードに割りふられたプロセスに割り当てる。カーネルレベルのスケジューラは非常に単純で効率的な優先度準拠のスケジューリングを提供する。それは単にレディ状態にある最も高い優先度のプロセスを取り上げ、それにプロセッサを割り当てる。一方、プロセスレベルのスケジューラは、カーネルの外で動作し、優先度を操作して、インタラクティブなプロセスとバックグラウンドのプロセスの間で効率的に時分割を実施する。V-System はリアルタイムプロセスに対して多くの高優先度レベルを予約することで特別な扱いを与えている。

プロセス移動

プロセス移動方針:V-System では、システムからの要求、あるいは、ユーザによる明示的な要求によって移動するプロセスを選択することができる。前者の場合、定期的に起動されるカーネルの手続きによって、プロセッサ間で移動するプロセスが選択される。これによって、カーネルの手続きは、プロセッサ間の処理負荷のバランスをとろうとする。一方、後者では、プログラムに対応するすべてのプロセスは次を起動することで一緒に移動される。

$$migrateprog[-n][< program - name >]$$

移動プロセスの適切な宛先ノードの選択は、遠隔プログラム実行の際、メタなマシン名 “*” を指定した場合のマシン選択と全く同様に行われる。指定したプログラムのプロセスは適切な宛先ノードが見つからない場合は移動されない。しかし、“-n” を指定した場合、移動できないなら、そのプログラムのプロセスは破棄される。更に、プログラム名が指定されていない場合は、このノードで実行されているすべての遠隔プログラムのプロセスを移動しようと試みる。

プロセス移動の機構

1. **アドレス空間転送の機構** (address space transfer mechanism):V-System はアドレス空間の転送に先行転送の機構を使っている、この機構については第 8 章で説明した。

2. **メッセージ転送の機構** (message-forwarding mechanism):3 つのタイプのメッセージ (タイプ 1、タイプ 2、タイプ 3) が移動するプロセスの宛先ノードへ転送される。この 3 つのタイプのメッセージについては 8 章で説明した。V-System はすべての 3 つのタイプのメッセージを転送するためにメッセージを再送する機構を使っている。この機構については 8 章で説明した。

3. **共プロセス処理機構** (mechanism for handing coprocess):V-System では、親プロセスが移動した場合、それに伴ってその子プロセスも移動することが保証されている。このために、親プロセスと子プロセスの分割を許さない方法が V-System のプロセス移動で使われている。

例外処理

V-System では、すべての例外が、**例外サーバ** (exception server) と呼ばれるユーザレベルのサーバプロセスによって処理される。例外条件が起きると、カーネルは問題を起こしたプロセスにその問題について記述したメッセージを例外サーバへ送させる。そして、例外サーバはそれを引き継ぎ、カーネルとその他のより高いレベルのサーバの機能を使ってその問題を扱うために必要な行動を開始する。この方法の主な利点は、カーネルの複雑性をほとんど必要することなく、強力で、柔軟で、ネットワーク透過な例外処理の機構の実装が可能であることである [Cheriton 1988]。

12.3.4 デバイス管理

V-Systemでは、デバイス管理の活動の多くは、ユーザレベルのサーバプロセスによって実行される。しかし、幾つかのデバイスサポートについてはカーネルによって提供されなければならない。なぜなら、デバイス割り込みはカーネルに行かなければならないし、幾つかのデバイス制御の操作は特権が与えられており、そして、幾つかのデバイス操作のカーネルによる制御は、カーネルの完全性に必要なものである [Cheriton 1988]。したがって、V-Systemのカーネルには、**カーネルデバイスサーバ** (kernel device server) モジュールがあり、これは、効率的に、信頼性良く、マシン非依存の形で、安全に広範なデバイスにアクセスできるように設計されている。カーネルデバイスサーバは、特定のデバイスに対しては最小限だけのデバイスサポートを提供しており、ユーザレベルのプロセスと個々のデバイスに対するドライバモジュール間のインタフェースであるデバイス非依存のコードを持っている。

カーネルデバイスサーバによって提供されるインタフェースは、**UIOインタフェース** (UIO interface) と呼ばれている。これによって、クライアントプロセスは、デバイスI/Oに対して標準的なI/O実行時サポートを使うことができる。また、ユーザレベルのサーバプロセスは、アプリケーションプログラマに対して、デバイスの拡張された抽象概念を実装することができる。例えば、カーネルデバイスサーバは基本的なブロックデバイスとして各ディスクデバイスへのアクセスを提供する。そして、ユーザレベルのファイルサーバはこの基本的なインタフェースを使ってファイルを実装する。同様に、ネットワーク通信に対して、カーネルデバイスサーバは、イーサネットパケットそのものを読み取り／書き込みする機能を提供することで、イーサネットのブロックインタフェースを提供する。そして、ユーザレベルのインターネットサーバはTCP/IP、UDP、X.25プロトコルを実装する。

デバイスI/Oは、読み取り／書き込みされるデータブロックの列として見えるUIOオブジェクトを作成することで達成される。UIOインタフェースはUIOオブジェクトに対して実行できる読み取り、書き込み、問い合わせ、修正操作の文法とセマンティクスを定義している。このサポートされている操作とブロック指向のデータアクセスモデルの使用によって、UIOインタフェースは、ディスク、テープ、プリンタ、ネットワークインタフェース、シリアルライン、ターミナル、マウスのような広範囲なデバイスを扱うのに十分に一般的なものとなっている。

12.3.5 プロセス間通信

プロセス間通信は V-System のカーネルによって提供される重要な機能である。プロセス間通信の機能の設計では特別な注意が払われており、RPC に類似したセマンティクスを使ってクライアントサーバ間で高速なメッセージ交換を提供するようになっている。カーネルによって提供されるプロセス間通信における 3 つの基本的な形式は、固定長のメッセージ転送による通信、データセグメントに対するアクセスを渡すことでの通信、そしてマルチキャスト通信である。これらについては次に説明する。

固定長のメッセージ転送

この形式の通信は、プロセスが固定長のメッセージを使って要求を送ったり、受け取ったり、応答したりするのに使われる。これを目的として提供されている 3 つのプリミティブは、Send、Receive、Reply である。Send プリミティブは、クライアントプロセスが手続きの引数と同値のものを渡すのに使われる。一方、Receive プリミティブと Reply プリミティブは、それぞれサーバプロセスがクライアントからの要求を受け取ったり、要求の実行結果をクライアントに返すのに使われる。すべてのメッセージは、厳密に同期的な方法で交換される。すなわち、メッセージを送ったプロセスは送られたメッセージが受信プロセスによって受け取られ応答を返されるまでその応答を待機しブロックしている。一方、クライアントの要求に応答した後は、サーバは Receive を実行し、てクライアントから送られてくるメッセージ待ちでブロックする。これらすべてのメッセージは固定長 (32 バイト) である。

データセグメントに対するアクセスの送信

この形式の通信では、プロセスは Send プリミティブを使ってメモリセグメント (連続したアドレス範囲) への擬似ポインタを固定長のメッセージとして受信プロセスへ送る。セグメントサイズとアクセスモードもこのメッセージの中で指定する。このメッセージを受け取ると、受信側のプロセスはこのセグメントをアクセスして、メッセージに指定されたアクセスモードに従って、読み取り／書き込みを行うことができる。この間、送信側は受信側からの応答を待っている。このような読み取りと書き込みはカーネルのプリミティブである CopyFrom と CopyTo によって処理される (図 12.6)。受信側は応答をする前にこれらのプリミティブを複数回実行することができる。更に、受信側は、セグメ

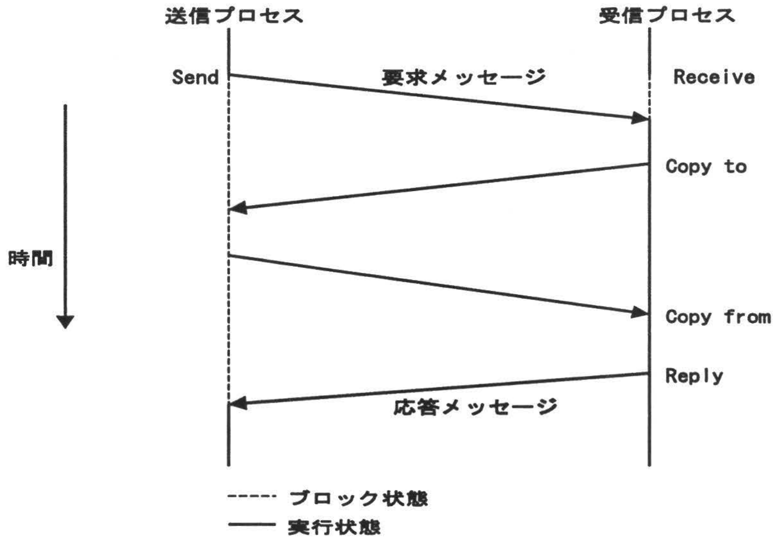


図 12.6 V-System におけるメッセージ交換の通信セマンティクス

ントへのアクセスと応答の権利を別のプロセスへ渡すことで、メッセージを転送することもできる。送信側は、送信したメッセージの応答を受け取るまでブロック状態にある。この機能によって、パラメータを参照で渡すことができる。

上記に述べた 2 つの形式でのメッセージ交換の通信セマンティクスを図 12.6 に示す。どちらの場合も、送信側と受信側は厳格に同期的な方法で相互作用する。更に、受信側は複数のメッセージを受信しキューイングすることができ、必要とするとき、そのうちの 1 つのメッセージに応答することができる。この柔軟性によって、メッセージ処理と応答に関して洗練されたスケジューリングを必要とするアプリケーションを書くことができる。

マルチキャスト通信

V-System のプロセス間通信の 3 番目の形式は、グループ (1 対多) の通信機能を提供する。V-System はプロセスグループ (又は単にグループ) の考えをサポートしている。プロセスグループはグループ識別子によって識別される一群のプロセスである。プロセスは同時に複数のグループに属することができ、自由にグループに参加したり脱退したりすることができる。

グループのメンバでないプロセスも含めて任意のプロセスは、Send のパラ

メータとしてプロセスの識別子ではなくグループの識別子を指定して、あるグループにメッセージを送信することができる。送信の後、送信側は送信したメッセージに対する応答のうち、0 個の応答か、1 個の応答か、あるいはそれ以上の応答を受け取るかを選択して、これによって実行をブロックすることもできるし、続行することもできる。0 個の応答は、グループに対する信頼性のないマルチキャストを実行するのに使われる。なぜなら、送信プロセスはメッセージを送信した後はブロックしないからである。1 つの応答はグループのうち少なくとも 1 つのメンバに対して信頼性のあるメッセージ配送を行う場合に使われる。なぜなら、送信側は 1 つの応答メッセージを受け取るまではブロック状態にあるからである。2 つ目以降の応答は他の幾つのプロセスがメッセージを受け取り、あるいは、応答を行ったかを示すことなく、単に破棄される。最後に、送信側が n 個の応答 $(n > 1)$ を選択受信する場合では、2 番目のメッセージから n 番目のメッセージまでのすべての応答メッセージが送信側のカーネルにキューイングされる。これによって、送信側は次のメッセージトランザクション、すなわち、次の Send の開始までにそれらを読み取ることができる。送信側がカーネルにキューイングされている後続の応答を受信するために GetReply というプリミティブが提供されている。送信プロセスは応答メッセージをどれぐらいの時間を待つかを決定できる。

グループへの Send では、メッセージは**限定子** (qualifier) も持つことができる。限定子は、宛先グループの中で限定子によって指定されたプロセスと共存するメンバに対してのみメッセージを配送すべきであることを示している。例えば、プロセス P を一時停止するために、一時停止要求を持つメッセージをプロセス P を指定する共存限定子と一緒にプロセス管理者のグループに送信することができる。これによって、要求メッセージはプロセス P の面倒を見ている管理者に対してのみ配送され、グループ全体には送られない。カーネルは単にその要求をプロセス P に対するホストアドレスにルーティングする。限定子は必ずしも 1 つのプロセスを指定する必要はなく、プロセスグループ識別子であっても良いことに注意してほしい。その場合は、限定子によって指定された宛先グループのすべてのメンバにメッセージが配送される。

マルチキャスト通信機能は V-System における多くの状況で使われている。例えば、各ノードのカーネルにあるタイムサーバとの時刻の同期情報を送信するのに使われており、また、分散スケジューリングの機構の一部として、負荷情報を配布するだけでなく、要求を行ったり、また、複製されたファイル更新のプロトコルでも使われている。

V-System のカーネルは注意深く構造化されており、通信操作におけるコストを最小化するようにしている。IPC の処理に関しては、カーネルは 2 つのモジュールから構成される。1 つはローカル IPC モジュールであり、もう 1 つはネットワーク IPC モジュールである。送信プロセスが Send 呼び出しを実行しメッセージを受信プロセスへ送信した場合、この呼び出しはカーネルにトラップされ、もし、受信プロセスがローカルな場合は、ローカル IPC モジュールによって処理される。もしそうでない場合は、それは VMTP を使ってネットワーク IPC モジュールによって処理され、遠隔にあるカーネルと遠隔にある受信プロセスと通信を行う。V-System における高速 IPC は比較的単純で基本的な IPC プリミティブによって達成されているだけでなく、VMTP を使うことにき依存している。要求／応答処理に関して最適化されている。VMTP は、マルチキャスト、データグラム、フォワーディング、ストリーミング、セキュリティ、そしてメッセージの優先度をサポートしている。VMTP の詳細は既に 2 章で説明したので、ここでは繰り返しを避ける。

12.3.6 V-System サーバ

V-System カーネルが最小の機能のみを提供し、従来の多くのカーネル準拠の機能をユーザレベルのプロセスとして実装するための基本的なフレームワークとして機能することをこれまでに見た。これによって、ユーザはアプリケーションに対して自分自身のサーバプロセスを設計し実装することができる。しかし、多くのアプリケーションによって共通に使われる幾つかのサービスがある。V-System はそのようなサービスに関するサーバを提供し、ユーザがそれらのサービスに対して自分自身のサーバを設計する必要がないようにしている。最も重要な V-System のサーバの幾つかが提供するサービスについて次に簡単に説明する。

プログラム管理者

プログラム管理者 (チームサーバ (team server) としても知られている) はプロセス管理の作業の多くを実行する。プログラム管理者は、従来の時分割のオペレーティングシステムでは通常はカーネルによって実行されている仕事を行う。プロセス作成、プロセス終了、プロセススケジューリング、様々なプロセスに対する資源消費の統計情報の管理などである。また、プロセス管理者はプロセスの優先度を操作する。これによって、インタラクティブプロセス、バッ

クグラウンドプロセス、ゲストプロセス (ある別のノードから来て、今はこのノードで実行しているプロセス) の間で効率的に時分割を実現することができる。更に、プログラム管理者はユーザレベルの例外処理としても機能し、障害を起こしたプログラムに対して対話型のデバッガを起動する。また、異なるノードのプログラム管理者は自動的な負荷分散機能とプロセス移動の作業のために協力する。

ファイルサーバ

ファイルサーバは従来のオペレーティングシステムではカーネルによって実行されていたファイル管理作業の多くを実行してくれる。特に、ファイルに読み取り／書き込みする形式でユーザがデータを格納あるいはアクセスすることができるように、物理的な記憶装置デバイスの抽象概念を提供している。ファイルサーバは通常多量なディスク記憶装置を持つ専用サーバマシンで動作している。V-System の多くのワークステーションはディスクレスであるため、ファイルサーバはネットワークでのクライアントからのファイルアクセス機能も提供している。

ファイルサーバは効率を目的としてマルチスレッド化されている。そのため、1 つのスレッドがディスクブロックを待ってブロックした場合、別のスレッドが実行できる。更に、ファイルサーバプロセスに属しているすべてのスレッドは共通のバッファキャッシュを共有している。このキャッシュは頻繁に使われるブロックをメインメモリ中に保持するために使われる。

インターネットサーバ

インターネットサーバはカーネルデバイスサーバによって提供される基本的なネットワークインタフェースデバイスの上に TCP/IP のツールを実現している。それはマルチスレッド化されており、効率化と柔軟性のためモジュール化した構造となっている。更に、より良い性能のために、リアルタイムスケジューリング、正確なタイミング、高速 IPC などの幾つかのカーネル機能を直接使っている。TCP/IP ツールは永久的に標準システムに組み込まれているわけではなく、必要なときにロードされる。

プリンタサーバ

プリンタサーバはプリントファイルをスプールする。一般に、プリントファイルは IPC 機能と UIO インタフェースを使ってプリンタサーバに送られる。

しかし、プリンタサーバがインターネットサーバの1インスタンスとして動作している場合は、プリントファイルはTCPコネクションを使って送ることもできる。

ディスプレイサーバ

ディスプレイサーバはビットマップディスプレイを使ってマルチウィンドウ機能を実現している。それはクライアントインタフェースにおいてハイレベルなグラフィカル表現を提供している。これによって、よく使われる操作はアプリケーションの機能に頼るのではなく、ディスプレイサーバにおいて実行されることができる。例えば、ディスプレイサーバはアプリケーションが利用できる機能として、複数ビュー、ズーム、再描画をサポートしている。

12.4 MACH

Machはマイクロカーネル準拠のオペレーティングシステムであり、Richard Rashidをリーダとして、DARPAの協力のもと、Carnegie-Mellon大学で開発された。それは以前にCMUで開発されたAccentと呼ばれるオペレーティングシステムをもとにしている [Rashid and Robertson 1981]。したがって、Machの多くの基本概念はAccentの研究結果に基づいている。しかし、Accentと比べて、Machは多くの改善された特徴を持っている。スレッドを利用したより細粒度の並行性、マルチプロセッサのサポート、より優れたプロセス間通信機構、より柔軟で効率的なメモリ管理手法などがこれには含まれる。

Machの最初の版は4CPUマルチプロセッサであるVAX 11/784をはじめとするDEC VAXコンピュータファミリに対して1986年にリリースされた。1987年までには、IBM RT PC、SUN 3、PERQ、Sequent、そして、Encoreが利用できるようになった。この時、Machはネットワークで相互接続されたマシン向けの分散オペレーティングシステムというよりは、共有メモリマルチプロセッサシステム向けのオペレーティングシステムであると考えられていた。なぜなら、Machを実行する多くのマシンは密結合の共有メモリマルチプロセッサシステムであったからである。疎結合の分散メモリマルチプロセッサシステムに適した拡張版 (Mach 2.5) がリリースされたのは後になってからであった。この版では大部分の4.2 BSD UNIX機能をMachカーネルに入れることで4.2 BSD UNIXとの互換性を提供した。大量のBSD UNIXコードがカーネルに存在したため、Mach 2.5のカーネルは非常に巨大にであり一枚岩のもの

であった。1989 年には、新版 (Mach 3.0) がリリースされた。ここでは、すべての BSD UNIX コードがカーネルから削除され、ユーザ空間へと移された。したがって、Mach 3.0 のマイクロカーネルが純粋な意味での Mach である。

1989 年に、コンピュータベンダのコンソーシアムである Open Software Foundation(OSF) が Mach を OSF/1 と呼ばれる最初のオペレーティングシステムの基本として選らんだ。OSF は IBM 社、DEC 社、HP 社などの主要な会社をメンバとしていた。NeXT ワークステーションもそのオペレーティングシステムとして Mach を使っている。

以下の Mach に関する説明は [Rashid 1987, Acccetta et al. 1986, Jones and Rashid 1986, Black 1990, Fitzgerald and Rashid 1986, Rashid et al. 1988, Coulouris et al. 1994, Tanenbaum 1995, Silberschatz and Galvin 1994] に基づいている。

12.4.1　設計の目標と主な特徴

Mach の設計は次に説明する研究と設計目標に影響を受けている。

オープンシステムアーキテクチャ

Mach の主な目標の 1 つは、新しいオペレーティングシステムを構築し、既存のオペレーティングシステムをエミュレートするための基礎を提供できるオープンなシステムを設計することであった。この目標を達成するために、Mach で用いられた設計哲学は、他の機能を作り出すために十分な基本的で小さい抽象概念のセットを提供する最小のマイクロカーネルを持つことと、ユーザレベルのサーバとして従来の多くのカーネルイキの機能を実装することである。このアプローチによって、UNIX や MS-DOS といった従来のオペレーティングシステムをオペレーティングシステムカーネルではなくアプリケーションプログラム、すなわち、特定のプログラミング抽象化をクライアントプログラムに提供するサーバあるいはサーバ群として考えることが可能であり、また、合理的でもある。Mach のマイクロカーネル準拠の設計によって、複数のエミュレータが同時に稼動することが可能であることに注意してほしい。これによって、同一のマシンで同時に UNIX と MS-DOS のような異なるオペレーティングシステムに対して記述されたプログラムを実行することができる。

BSD UNIX との互換性

ごく初期のころから、Machの重要な目標は、BSD UNIXとのフルの互換性を提供することであった。これによって、研究や大学のコミュニティで広く受け入れられる。この目標ははじめはMachと4.2 BSD UNIXを1つのカーネルにすることで達成された。しかし、4.2 BSD UNIXとの完全な互換性を保証する一方で、カーネルの肥大化を招いた。オープンシステム設計の戦略は後になって適用されてた。そして、BSD UNIXのコードはカーネルからすべて取り除かれ、ユーザ空間に押し込められた。この再設計によって、Machのカーネルは純粋にMachからなる最小のマイクロカーネルに軽量化され、BSD UNIXだけでなくその他の既存のオペレーティングシステムをエミュレートする基本として、また、新しいオペレーティングシステムを設計する基本として使うことができるようになった。

ネットワーク透過性

他の分散オペレーティングシステムと同様に、ネットワーク透過性はMachの目標でもあった。この目標を達成するために、より高いレベルのネットワークワイドな名前付けシステムが使われている。それは、**ネットワークメッセージサーバ** (network message server) と呼ばれるユーザレベルのサーバとして実装されており、システムの異なるノードに配置されている2つのプロセス間で、透過的なメッセージ送信を行うのに使われている。この機能によって、ネットワークワイドな資源に対する透過的なアクセスが可能となる。

柔軟なメモリ管理

Machのもう1つの重要な目標は、強力で柔軟なメモリ管理システムをサポートすることであった。この目標を達成するために、Machは手の込んだ仮想メモリシステムを提供しており、これは固定サイズのページで実現されている。Machのメモリ管理システムの魅力的な点かは次の通りである。

1. メモリ管理システムのマシン非依存の部分とマシン依存の部分を明確に分けている。これによって、メモリ管理システムを他のシステムのものよりかなり移植可能なものとしている。

2. 通信システムと統合され、高速なローカルIPCの実現を可能としている。

3. 2つ又はそれ以上のプロセスで効率的にデータ共有を行うために、書き

込み時コピー (copy-on-write) の機構を持っている。この機構では、データは変更があった時にはじめて物理的にコピーされる。したがって、カーネル、ブロック I/O デバイス、クライアント、サーバ間での大量のデータ移動を潜在的に削減できる。

4. 継承 (inheritance) の機構を持っている。これによって、親プロセスはどのメモリ領域が子供によって継承されるべきかを宣言することができ、また、どのメモリ領域が読み取り／書き込み可能であるべきかを宣言することができる。この機構によって、親プロセスと子プロセスの間で保護を行うための様々な共有方策を提供することができる。

5. 外部メモリ管理者 (external memory manager) という概念を持っている。これによって、複数のユーザレベルのメモリ管理者を実装し使うことができる。各ユーザレベルのメモリ管理者は自分が支援しているオブジェクトに適した自分自身のセマンティクスとページングアルゴリズムを実装することができる。外部メモリ管理者という概念はページ準拠の分散共有メモリシステムを実装するのにも適している。

柔軟な IPC

Mach のもう 1 つの目標は、柔軟な IPC システムを提供することであった。これによって、プロセスは信頼性のある効率的な方法で通信を行うことができる。この目標を達成するために、Mach はポートに基づくメッセージ準拠の IPC を使っている。ポートはメッセージを保持するカーネルオブジェクトである。この IPC システムは次の特徴を持っている。

1. 同期および非同期メッセージパッシングをサポートする。

2. 信頼性のある、そして順序付けられたメッセージ配送を保証する。

3. ポートに対するアクセスを制御するのに、ケーパビリティに基づくアクセス制御機構を使っており、これによって安全なメッセージ通信が保証される。すべてのメッセージはポートへ送られ、またポートから受信される。

4. ネットワーク透過性をサポートしている。これによって、受信側の位置を知る必要なく、送信側はメッセージを他のノードにある受信側へ送ることができる。

5. メッセージの送信側と受信側が同じノードにある場合、何らコピーをする

必要なく、大量のデータを転送する方法を提供する。

6. 異なるタイプのマシン間でデータ転送を行う際、一方のデータ表現から他方のデータ表現へデータタイプを変換することで異機種性をサポートする。

高性能

高性能であることはMachの目標でもあった。しかし、マイクロカーネル準拠の設計によるアプローチを用いることは性能上で問題が生じやすい。なぜなら、サービスプロセスとマイクロカーネルとのメッセージパッシングにはコンテキスト切り替えが必要であり、システムを低速化する。高性能を実現するために、Machの設計において使われた主な技法は次の通りである。

1. マルチスレッド化されたプロセスを使う。これによって、並列処理向けの細な粒度の並列性という利点を活用することができる。

2. 高速なローカルIPC向けの制御権移行型のスレッドスケジューリングを使う。例えば、クライアントはサーバに対して制御権を移行することができ、サーバはローカルIPCが完了した時点で制御権を戻す。このアプローチによって、コンテキストスウィッチを避けることはできないが、スケジューラを通るパスを最適化あるいは避けるとができる。

3. 書き込み時コピーの機構を使ってデータのコピーを最小化する。従来のカーネルにおける多くの操作のうち最もCPUコストが高いのはデータのコピーである(例えば、バッファのイン、アウト)。Machの書き込み時コピーの機構を使うことで、データコピーのコストを大きく削減することができる。

4. ユーザ空間の透過的な共有ライブラリを使って、クライアントアドレス空間でサーバの作業を実行する。Machでは、多くのカーネルの動作がカーネルから直接にクライアントのアドレス空間に効率的に移される。これは、透過的な共有ライブラリに動作を置くことで達成される。この方法によって、可能な限り、サーバの要求はライブラリの中で実装され、メッセージあるいはカーネルトラップの必要性を避けることができる。

単純なプログラミングインタフェース

Machのもう1つの目標は、プログラマに対して単純なインタフェースを提供することであった。この目標を達成するために、MachはMach Interface Generator(MIG)と呼ばれるインタフェースジェネレータを提供している。MIGは基本的にはサービスの定義からスタブ手続きを生成するコンパイラである。

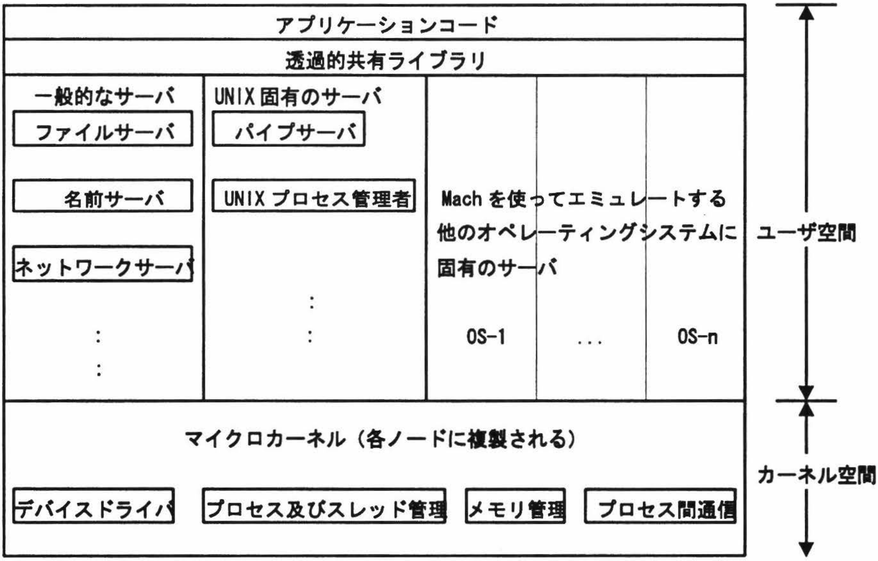


図 **12.7**　Mach システムアーキテクチャ

すべてのサービスに対してスタブ手続きは透過的な共有ライブラリに置かれている。マニュアルで説明され、アプリケーションプログラムから呼ばれるのは、まさにこの手続きである。このアプローチによって、アプリケーションプログラマは単に手続き呼び出しを行うだけでサービスを使うことができる。システム呼び出しを実行したり、メッセージを送受信するためのコードを書く必要はない。

12.4.2　システムアーキテクチャ

図 12.7 に示すように、Mach のシステムアーキテクチャは主に次の層から構成されている。

1. **マイクロカーネル層**:最も低い層は最小のマイクロカーネルである。これはシステムの各ノードに複製されている。これは最小である。なぜなら、他のすべての機能と特徴を導き出すのに十分に強力であり充実した基本的な抽象概念の小さな集合から構成されているからである。特に、この層は IPC、メモリ管理、プロセス管理、I/O サービスと関連している。これらの機能を実行する

ために、5つの基本的な抽象概念を提供している。すなわち、タスク、スレッド、メモリオブジェクト、ポート、メッセージである。タスクとスレッドはプロセス管理の抽象概念であり、メモリオブジェクトはメモリ管理で使われる抽象概念である。また、ポートとメッセージはプロセス間通信の抽象概念である。

2. **ユーザレベルのサーバ層**:Machでは、従来の多くのカーネルの機能がユーザレベルのサーバとして実装されている。これらのサーバはシステム全体のアーキテクチャの次の層を形成している。この層のサーバは大きくは2つのカテゴリに分類される。すなわち、汎用と特殊である。**汎用サーバ** (generic server) は一般的な事柄に対するサービスを提供する。例えば、ユーザ認証、分散ファイル管理、オブジェクトの透過的名前付け、ネットワークプロトコル、デバイスア住置などのサービスに対して、汎用サーバがある。一方、**特定サーバ** (specific server) はあるノードにおいて特定の動作を提供するために使われる。例えば、UNIXのような既存のオペレーティングシステムの機能を提供するために、一連の特定サーバを使うことができる。図12.7に示すように、特定サーバの複数のセットが、同じノードで異なるオペレーティングシステムの機能をエミュレートするために使われている。

3. **透過的な共有ライブラリ層**:この層はすべてのサービスに対してMIGによって生成されたスタブ手続きを含んでいる。アプリケーションプログラマは単に手続き呼び出しを実行するだけでサービスを使うことができ、システム呼び出しを実行したり、メッセージ送受信のためにコードを書いたりする必要はない。

4. **アプリケーションコード層**:この層はアプリケーションプログラムのコードを含んでいる。

12.4.3 プロセス管理

基本抽象概念

Machでプロセス管理に使われている2つの基本抽象概念はタスクとスレッドである。これらは基本的には第8章で説明した“プロセス”と“スレッド”の概念に対応している。すなわち、**タスク** (task) は実行環境であり、**スレッド** (**thread**) は実行の基本単位である。資源はタスクに対して割り当てられ所有

される。タスクに属するすべてのスレッドはその資源を共有する。混乱を避けるために、Mach に関する以降の説明では「プロセス」という言葉を「タスク」の代わりに使うことにする。

プロセスとスレッドの状態

任意の時点で、スレッドは次の状態のいずれか 1 つの状態にある。

1. **実行中** (running):実行中のスレッドはあるプロセッサで実行を行っているか、実行する条件は整っているがプロセッサが割り当てられるのを実行キューで待っているかのいずれかである。カーネルでブロックしているスレッド (例えば、ページフォールトが満たされるのを待っている) もまた実行状態にあると考えられる。

2. **停止中** (suspended):停止中のスレッドは、プロセッサで実行もされてなければ、プロセッサの割り当てを実行キューで待ってもいない。このスレッドは実行状態に戻されるまで実行を行わない。

スレッドと同様に、プロセスもまた実行状態と停止状態とがある。プロセスはスレッドの状態とは独立に、実行中あるいは停止中になる。しかし、プロセスの状態はそのプロセスに属するすべてのスレッドに影響する。したがって、プロセスが停止状態になると、それに属するスレッドは現在の状態に関係なく実行できない。すなわち、スレッドはそれ自身とそれが属するプロセスの両方が実行状態にある時のみ実行できる。

プロセスとスレッドはプログラムの制御によって停止したりそして再開したりできる。このため、各プロセスとスレッドに付随して**一時停止** (suspend) カウンタがある。process_suspend と thread_suspend というプリミティブが提供されており、一時停止カウンタをインクリメントする。また、process_resume と thread_resume をそれをディクリメントする。スレッド (又はプロセス) の一時停止カウンタが正になると、停止される。そして、一時停止カウンタが 0 になると、状態は実行中になる。この機構によって、スレッド (プロセス) に対して複数の一時停止呼び出しが実行できる。そして、再開の呼び出しが同じ数だけ行われたときだけ、スレッド (あるいはプロセス) は再開される。したがって、カウンタを使うことによって、ビットを使うよりもより柔軟であり、競合条件を避けるのに役立つ。

プロセスとスレッドに関する操作

各プロセスはプロセスポート (process port) を所有しており、各スレッドは**スレッドポート** (thread port) を所有している。これらのポートは、プロセスとスレッドを操作するための機構を提供する。特に、スレッド (あるいはプロセス) に対する操作は、メッセージをスレッドポート (あるいはプロセスポート) に送ることで起動される。プロセスポートに対するアクセスは、間接的には、そのプロセスに属するすべてのスレッドへのアクセスを許可している。しかし、逆はそうではない。

Machが提供するプロセス管理のプリミティブには、プロセスの作成、プロセスの消滅、プロセスの停止と再開、どのスレッドがどのプロセッサあるいはプロセッサグループで実行できるかを制御したり、現在あるいは将来のスレッドのスケジューリングに対してプロセスの優先度を設定したり、プロセスのすべてのスレッドのリストを入手したり、プロセスの統計的な情報を入手するなどのプリミティブがある。

Machでは、スレッドはカーネルによって管理される。すなわち、スレッドの作成と廃棄はカーネルによって行われ、カーネルのデータ構造の更新を起動する。基本的なカーネルのインタフェースはスレッドの操作に対して様々なプリミティブを提供している。それらは、1つのアドレス空間での複数の動作を処理するための機構を提供する。しかし、プログラマにこのような低レベルのプリミティブによって作業を行ってもらうより、MachはCとその他の言語による多くのより高レベルのプログラミングインタフェースを提供している。そのようなインタフェースの1つがC-threadsパッケージ (C-threads package) である。その概要は次に説明する。

C-threadsパッケージ

C-threadsパッケージによって、プログラマは簡単で便利な方法でカーネルのスレッドプリミティブを使うことができる。C-threadsパッケージはカーネルインタフェースのすべての能力を提供しているわけではないが、標準的なプログラマにとっては十分に良いものである。特に、C-threadsパッケージにはスレッドを直接に制御するルーチン、スレッドの臨界領域に対して相互排他を強制するためのルーチン、スレッドの一般的な同期を行うためのルーチンがある。

直接のスレッド操作のルーチンを以下に示す。

■ cthread_fork は呼び出したスレッドと同じアドレス空間に新しいスレッド

を作成するのに使用する。スレッドはその親スレッドと並行に実行を行う。しかし、親のコードを実行する代わりに、このルーチンの入力パラメータに指定された手続きを実行する。

■ cthread_exit はこれを呼び出したスレッドを終了するのに使われる。このルーチンはスレッドの作業が終わったときに、スレッドから呼び出される。

■ cthread_join は指定した子スレッドが終了するまで、呼び出したスレッドが自分自身を停止するのに使われる。

■ cthread_detach は指定したスレッドが決して cthread_join されない (待たれない) ことを表明するために使われる。そのスレッドが cthread_exit を呼び出した場合、そのスタックと他の状態に関する情報はすぐに削除される。通常、この除去は親が cthread_join に成功した後で実行される。

■ cthread_yield は何もすることがなく自発的にスレッドが CPU を放棄しスケジューラへ渡す時に使われる。スケジューラは別のスレッドをスケジュールして CPU で実行する。

プロセスに属するすべてのスレッドは共通のアドレス空間を共有しているので、臨界領域がスレッドによって実行されるとき、相互に排他的に実行されなければならない。Mach では、ミューテック変数の技法がこの目的のために使われている。関連する C-threads パッケージルーチンを次に示す。

■ mutex_alloc がミューテック変数を動的に割り当てるのに使われる。

■ mutex_free がミューテック変数を解放するのに使われる。

■ mutex_lock はミューテック変数をロックするのに使われる。ミューテック変数が既にロックされている場合、スレッドは成功するまでロックを試みる。ロックを持っているスレッドが同じミューテックに対してロックを試みる場合、デッドロックを招くことに注意してほしい。C-threads パッケージは上限のある待機を保証していない。というより、それはミューテックルーチンを実装するために使うハードウェア命令に依存している。

■ mutex_unlock はミューテック変数をアンロックするのに使われる。

最後に、Mach では条件変数の技法がスレッドの一般的な同期に使われている。関連する C-threads パッケージのルーチンを次に示す。第 8 章を思い出してほしい。条件変数はミューテック変数と関連付けられている。

■ condition_alloc は条件変数を動的に割り当てるのに使われる。

■ conditon_free は割り当てられた条件変数を削除するのに使われる。

■ condtion_wait は条件変数に関連づけられているミューテック変数をアンロックするために使われる。また、呼び出しスレッドをブロックする。

■ condition_signal は条件変数でブロックしているスレッドに、待っているイベントが起きたことを示すために使われる。その後、関連するミューテック変数はロックされ、通知されたスレッドは臨界領域で実行を開始する。condition_signal は、ブロックが解けたスレッドが condtion_wait 呼び出しから最後に返ってきたときに、直前の条件が成り立っていることを保証していない。したがって、起こされたスレッドはそのブロックがとけ、なおかつ条件が成り立つまで、condition_wait ルーチンをループで実行しなければならない。

スレッドスケジューリング

Mach のスレッドスケジューリングの方式を図 12.8 に示す。**プロセッサ集合** (processor set) の概念が使われており、ここでは、システムのすべてのプロセッサがソフトウェアによってばらばらの集合にグループ化されている。スレッドの計算要求と各プロセッサの負荷に依存して、各スレッドはソフトウェアによってプロセッサ集合に 1 つに割り当てられる。このように、各プロセッサ集合には CPU の集合とスレッドの集合が存在する。スケジューリングを目的として、プロセッサ集合の CPU とスレッドは他のプロセッサ集合とは完全に独立になっている。すなわち、スケジューリングアルゴリズムは主にプロセッサ集合のスレッドを公平で効率的な方法でプロセッサ集合の CPU に割り当てることに関わっている。公平さと効率化のため、スケジューリングアルゴリズムは動的に可変なクオンタムによる優先度準拠の方法を使用している。そ

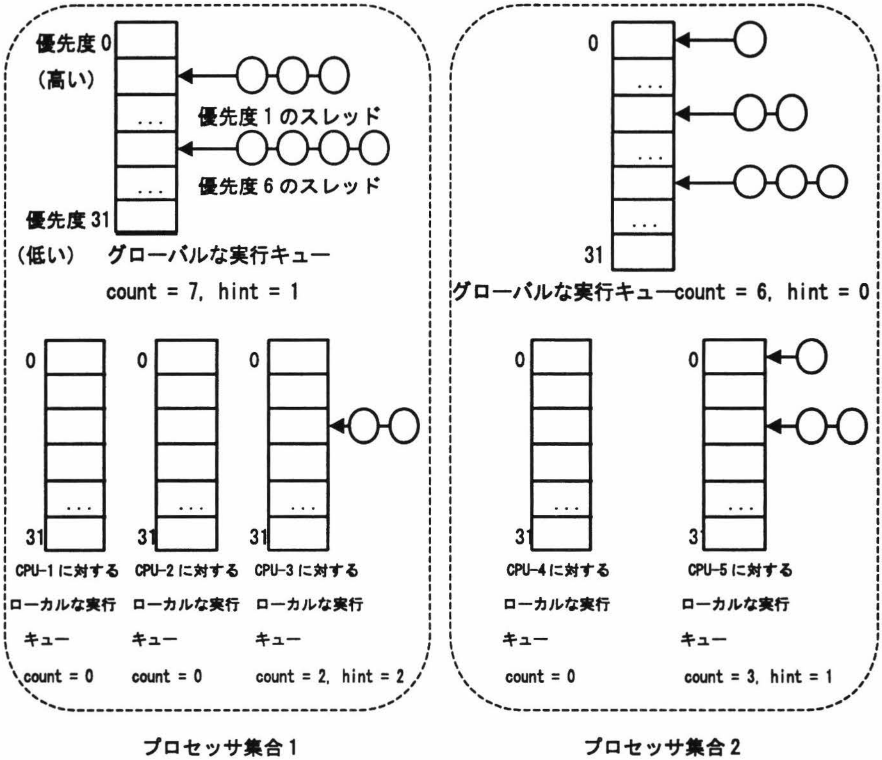


図 **12.8**　Mach におけるスレッドスケジューリング

の詳細を次に与える。

Mach では、各スレッドは 0 から 31 の優先度が関連づけられている。0 は最も高い優先度であり、31 は最も低い優先度である。更に、図 12.8 に示すように、各 CPU に関連づけられた**ローカル実行キュー** (local run queue) と、各プロセッサ集合に関連つけられた**グローバル実行キュー** (global run queue) とがある。これらのキューのそれぞれは 32 のキューの配列であり、それぞれのキューは 0 から 31 の優先度に対応している。ローカル実行キューとグローバル実行キューの両方が関連する 2 つの変数、カウントとヒントを持っている。カウント変数には実行キューの 32 すべてのキューにあるスレッドの数が格納されている。ヒント変数には 32 のキューの中で最も優先度が高いスレッドを現在持っているキューの番号が格納されている。ヒント変数によって、最も高い

空のキューを検査することなく、最も高い優先度を持つスレッドの検索が効率的に行える。これらの2つの変数に加えて、各グローバル実行キューは関連するミューテック変数を持っている。これはグローバル実行キューをロックして、同時には1つのCPUだけがそれを操作することを保証するために使われる。

プロセッサ集合のグローバル実行キューはプロセッサ集合にあるCPUのどれかに割り当てることができるスレッドを保持している。一方、各ローカル実行キューは、対応するCPUに永久に束縛されたスレッドを保持している。例えば、個々のCPUに接続されたデバイスのデバイスドライバであるスレッドは、そのCPUでのみ実行しなければならない。そのようなスレッドをグローバル実行キューに入れるのは間違いである。なぜなら、その場合、同じプロセッサ集合の別のCPUによってそのスレッドが取り上げられ実行されるかもしれないからである。特定のCPUについては、ローカル実行キューにあるスレッドはグローバル実行キューにあるスレッドよりも高い優先度を持っている。

プロセッサ集合におけるキューとスレッドのこのような取り合わせによって、基本的なスケジューリングアルゴリズムは次のように動作する。

1. 優先度nのスレッドの状態が**実行中** (running) になると、(もし、永久にCPUに束縛されるのなら) 特定のCPUのローカル実行キューかグローバル実行キューのキューnの最後に置かれる。実行状態にないスレッドはどのキューにも存在しないことに注意してほしい。

2. CPU上で現在実行中のスレッドが、ブロック、終了、CPU放棄、あるいはクオンタムを使い果たした場合、CPUは最初にローカル実行キューを検索し最も高い優先度のスレッドを見つける。このため、最初にカウント変数を検査する。もし、それがゼロでない場合、ヒント変数を取り出し、最も高い優先度のスレッドを見つけるためにその番号を持つキューを検索する。もし、ローカル実行キューが空の場合 (カウント変数の値が0であると分かった場合)、CPUは同様にしてグローバル実行キューの検索を行う。しかし、今度は、検索を行う前にグローバル実行キューを最初にロックする。

3. 実行可能なスレッドがどちらにもない場合、何らかのスレッドの実行準備が整うまで、特別な遊休スレッドが実行される。

4. 一方、CPUが実行可能なスレッドを見つけた場合、1クオンタムの間、

そのスレッドを実行する。クオンタムが終了すると、ちょうど実行しているスレッドよりも優先度が高いか、又は同じ優先度を持つ別のスレッドが実行可能かどうかの検査が行われる。この検査はローカルとグローバル実行キューの検索を含む。そのようなスレッドが見つかった場合、現在のスレッドから CPU が取り上げられ、そのスレッドに割り当てられる。そして、現在のスレッドは適切な実行キューの優先度キューの最後に置かれる。もし、そのようなスレッドがない場合は、同じスレッドはもう 1 クオンタム実行される。

上記に説明した基本的なスケジューリングアルゴリズムに加えて、Mach のスケジューリング方式は他の重要な特徴がある。

1. システムの負荷が軽いときは高い効率性を提供し、システムの負荷が高いときは小さいジョブに対して応答を良くするために、時間の**クオンタム長**は実行可能な全スレッド数に反比例して変わる。

2. スレッドもまた先取り的にスケジュールするができる。例えば、プロセッサ集合の負荷が突如として上がることでクオンタム長が突然として小さくなった場合、現在実行中のスレッド (それは既に正当な長いクオンタム長を与えられている) は横取りされる。なぜなら、それに割り当てられたクオンタムは十分に利用したからである。この特徴によって、システムの負荷が突然大きくなった状況における公平な CPU 利用が可能となる。スレッドはもし現在 CPU を使うことがないなら、クオンタムが切れるまえに CPU を自発的に明け渡すこともできる (cthread_yield ルーチン)。

3. スレッドの優先度は動的に変更される。これによって、最も高い優先度をもつスレッドによって CPU が独占されることを防ぐ。この特徴によって、利用可能な計算能力がすべてのスレッドに公平に利用できるようになる。

4. ハンドオフスケジューリング機能を使って、実行キューをバイパスして、直接的に現在実行中のスレッドによって指定されたスレッドに CPU をスイッチすることができる。

5. マルチプロセッサでは、Mach は密関係スケジューリングを実行するようにも構できる。

Machのスレッドスケジューリングに関する更なる詳細は [Black 1990] に見ることができる。

例外処理

Machの例外処理機能では、例外の使用は次のカテゴリに分類される。

1. 誤り処理
2. デバッグ

例外のこの2つのクラスは異なる方法で処理される。前者のクラスに属する例外は、スレッドごとに基づいて処理される。一方、後者のクラスに属する例外は、**例外ポート** (exception port) と呼ばれるそれに関連づけられた特別なポートをもつ、1つの例外ハンドラによって処理される。例外がおきると、カーネルは適切な例外ハンドラに対して例外を記述したメッセージを送信する。2つのタイプの例外は異なる方法で処理される。なぜなら、1つのスレッドのみをデバッグしようとすることにはほとんど意味がないし、あるいは、複数のデバッガを起動して複数のスレッドからの例外を受けることにもほとんど意味がないからである。

デバッグをする際、プロセスの例外処理機能がその子プロセスによって継承される。これによって、デバッガはプロセスの木全体を操作することができる。一方、誤り処理では、プロセスの例外処理機能は子プロセスによって継承されない。デフォルトの仮定は、プロセスは例外処理機構を持たないということである。

Machでは、両方のタイプの例外が同時に起きた場合、誤りハンドラがデバッガよりも優先される。これは、誤りハンドラは通常はプロセスの一部であり、したがって、デバッガが存在したとしても通常どおりに実行されるべきであることによる。

12.4.4 メモリ管理

仮想メモリ管理

Machでは、メモリはページングによってサポートされた線形な仮想アドレス空間としてモデル化されている。各プロセスは自分自身の仮想アドレス空間

を持っており、そこでスレッドは実行する。プロセスのアドレス空間は一般的にはまばらであり、使われている仮想アドレス空間のセクションの間には未割り当ての空間による穴がある。これは、新しいアイテムがアドレス空間にマッピングされたり、アドレス空間から削除されるごとに、Mach はアドレス空間を圧縮して未割り当てのメモリの穴を削除しようとはしないからである。

各プロセスの仮想アドレス空間を管理するためによく使われてきた方法は、カーネルの中に 0 から使用される最大のページ番号までのリニアなページテーブルを保持することである。Mach のプロセスの仮想アドレス空間はまばらであるので、適度なページサイズに関して、このアプローチはかなり大きなページテーブルに対応する大量のメモリが必要になるであろう。そして、これは高価なものとなる。この問題を克服するために、Mach では**リージョン** (region) の概念が使われている。リージョンとはベースアドレスとサイズによって表現されたアドレス空間の連続領域である。リージョンはページ境界に沿っていなければならない。プロセスの仮想アドレス空間を管理するために使われるテーブルのサイズは、それに対して現在割り当てられているリージョンについてのみの情報を保管することで、管理可能なサイズになっている。仮想アドレスは、割り当てられているリージョンに帰着する場合にのみ有効である。未割り当てのリージョン (使用されていない仮想アドレス空間) にあるアドレスへのアクセスは、トラップを生じる。しかし、プロセスが望むならば、そのトラップを捕まえることもできる。

Mach は仮想アドレス空間を操作するために多くの呼び出しを提供している。例えば、vm_allocate 呼び出しは仮想メモリの新しいリージョンを割り当てる。呼び出し側は、ベースアドレスとサイズの両方か、サイズのみを指定することができる。前者の場合、システムは指定されたリージョンを割り当てる。一方、後者の場合、システムは適当なリージョンを探し割り当て、呼び出し側にベースアドレスを返す。vm_deallocate 呼び出しはプロセスのアドレス空間からリージョンを削除し、そのリージョンをもはや有効でないものにする。vm_read と vm_write 呼び出しによって、プロセスが別のプロセスの仮想メモリにアクセスすることができる。vm_copy によって、メモリリージョンが新しいリージョンにコピーされる。これらの呼び出しに加えて、メモリリージョンのアクセス保護を制御したり、新しいプロセスが作成されるとき親プロセスから子プロセスへのメモリリージョンの継承を制御したり、プロセスのアドレス空間にあるリージョンについての情報を取得するような目的のために使う呼び出しきある。

外部メモリ管理者

従来のオペレーティングシステムにおいて、二次記憶装置(例えばディスク)の管理は通常カーネルが供給するファイルシステムに基づいて行われている。このファイルシステムは、ページング方式、共有セマンティクス、対応するオブジェクト(例えばファイル)の実装詳細について決めている。そのようなオブジェクトを管理するためにシステムに組み込まれた方式は、多種のオブジェクトには適したものであるが、すべてのオブジェクトタイプに対しては最適なものでは限らない。したがって、Machのメモリ管理システムはユーザが自分自身の特別な目的を持ったメモリ管理者を実装し使うことができるように設計されている。そして、ユーザの自身のメモリ管理者では、特別な要求を持つオブジェクトに対して独自のメモリ管理方式を持つことができる。このような特別な目的を持つ、ユーザレベルのメモリ管理者は**外部メモリ管理者** (external memory manager)と呼ばれている。

外部メモリ管理者の考えを実現するために、Machのメモリ管理システムで使われている重要な抽象概念は、**メモリオブジェクト** (memory object)である。メモリオブジェクトは一連の操作(例えば、読み取りと書き込み)が定義されているデータの集まりを表現した抽象オブジェクトである。また、メモリオブジェクトはプロセスの仮想アドレス空間の未使用の部分にマッピングすることもできる。これによって新しいリージョンを形成する。

メモリオブジェクトは外部メモリ管理者によって作成され管理される。図12.9に示すように、プロセスのアドレス空間にマッピングされたそれぞれのメモリオブジェクトに対して、それを制御する外部メモリ管理者が存在する。Machのカーネルは単にこれらのメモリオブジェクトに対するキャッシュ管理者として働く。カーネルは、他の仮想メモリ実装と同様に、マッピングされているすべてのメモリオブジェクトのメモリ中に存在するページのキャッシュを管理する。しかし、外部メモリ管理者はユーザレベルのプロセスであるので、異なるクラスのメモリオブジェクトを処理する異なるメモリ管理者を持つことができる。これによって、一連の異なる操作、異なる共有セマンティクス、メモリオブジェクトが異なるクラスのメモリオブジェクトに対してマッピングされた後オブジェクトがどうなるかに関する異なるルールを柔軟に提供することができる。この柔軟性によって、ユーザが記述したメモリ管理者が管理するメモリオブジェクトに対して自分自身のページング方式を持つこともできる。各メモリ管理者はメモリ中にないページを自分自身のどこへ格納すればよいか、また、メモリオブジェクトが破棄されたとき変更のあったページを二次記憶に

書き戻すかどうかについて決定することができる。メモリオブジェクトは、その内容や重要性に関して Mach が仮定を持たないという意味において、カーネルとは独立なものである。

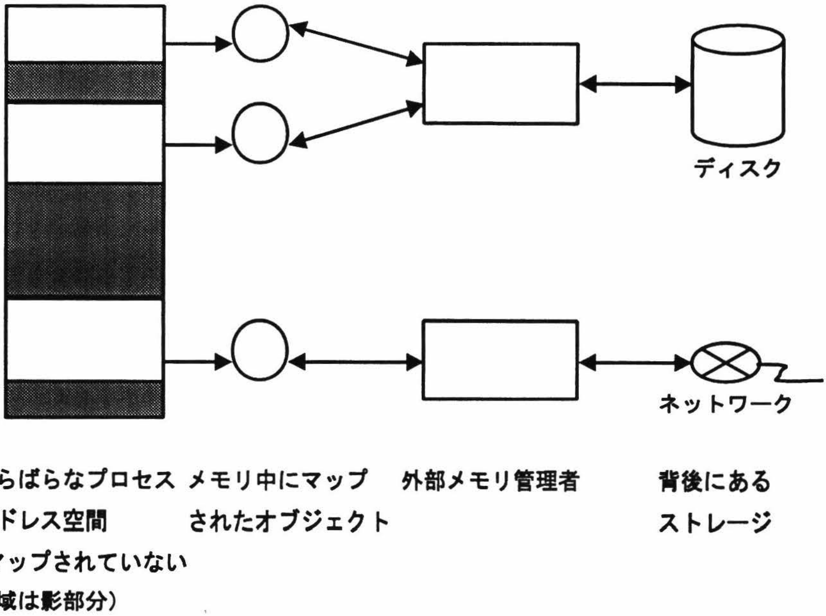


図 **12.9**　外部メモリ管理者とメモリオブジェクト

ユーザが自分自身のメモリ管理者を記述し使うことを可能にするために、Mach はカーネルとメモリ管理者の間に良く定義されたインタフェースを提供している。このインタフェースは、カーネルとメモリ管理者の間で交換することのできる一連のメッセージタイプから成る、通信プロトコルの型をとっている。これらのメッセージタイプは 2 つのカテゴリに分類される。つまり、カーネルがメモリ管理者へ送るものと、メモリ管理者がカーネルへ送るものである。カーネルからメモリ管理者へ送られる幾つかの例を次に挙げる。

■ memory_object_init: このメッセージはメモリオブジェクトが最初にマッピングされたときに送られる。メモリ管理者はこのメッセージを受け取ると、自分自身を初期化する。メッセージには、マッピングされるメモリオブジェクトに対する制御ポートと名前ポートが含まれる (これらのポートタイプについ

ては後で説明する)。これらのポートはそのメモリオブジェクトに関連するメッセージをカーネルへ送るためにメモリ管理者によって後ほど使われる。

■ memory_object_data_request: このメッセージはメモリオブジェクト上でページフォルトが起きたとき、メモリ管理者が管理するメモリオブジェクトのデータを要求するために送られる。このメッセージには要求データの範囲が入っている。

■ memory_obuject_data_write: このメッセージはカーネルがメモリ中から1つ又は複数のメモリオブジェクトのダーティページを削除する必要があるときに送られる。例えば、ページが古くなることでそのような場合が起きる。二次記憶に書かれるべき更新データはこのメッセージ中に含まれる。

■ memory_object_data_unlock: このメッセージはメモリオブジェクトのメモリ管理者に対して別のプロセスが使えるようにオブジェクトのロックされたページをアンロックするように要求するために送られる。

■ memory_object_lock_completed: このメッセージはカーネルによって実行された (以下で説明する)memory_object_data_lock 要求に対する応答としてメモリ管理者に送られる。

■ memory_object_terminate: このメッセージはメッセージで名前を指定されたメモリオブジェクトがもはや使用しておらず、メモリから削除することができるということ通知するために、メモリ管理者へ送られる。

メモリ管理者からカーネルへ送られるメッセージタイプの例を次に挙げる。

■ memory_object_set_attributes: このメッセージはカーネルから受け取った memory_object_init に対する応答として送られる。このメッセージは、カーネルに対して、メモリ管理者が新しくマッピングされたオブジェクトに対する要求を受け付ける準備ができたことを示す。

■ memory_object_data_provided: このメッセージは memory_object_data_reeequest メッセージの応答として送られ、カーネルに対して要求されたページを返す。

■ memory_object_data_unavailable: このメッセージは memory_object_data_reques メッセージの応答として要求データが利用できないときに送られる。

■ memory_object_data_lock: このメッセージはカーネルに対してページの保護モードを変更するよう要求するために送られる。このメッセージには、指定されたデータに対する新しい保護モード (読み取り、書き込み、実行) を指定した lock_value が含まれている。

■ memory_object_data_clean: このメッセージはメモリ管理者がメッセージで指定したページをディスクに書くことでクリーンにすることができるように、カーネルに対してそのページを送るように要求する。

■ memory_object_data_cache: このメッセージはどのプロセスもアドレス空間にマッピングしていないにも拘わらずメモリオブジェクトのキャッシュデータを保持するかどうかをカーネルに伝えるために送られる。

■ memory_object_destroy: このメッセージはあるメモリオブジェクトがもはや必要なく、カーネル中の情報は破棄することができる、ということをカーネルに通知するために送られる。

カーネルからメモリオブジェクトへのすべてのメッセージは非同期に送られる。なぜなら、応答することのできないユーザプロセスを待っているスレッドでカーネルがブロックすることは理にかなっていないからである。カーネルとメモリ管理者で交換されるメッセージは、Mach のポート準拠の通信機構 (後で説明する) を使って交換される。このために、次のタイプのポートが使われる。

1. **オブジェクトポート** (object port):各メモリオブジェクトには関連するオブジェクトポートがある。このポートはオブジェクトを管理するためにメモリ管理者によって作成される。メモリ管理者はこのポートを使って、ページフォルトやその他のオブジェクトに関係するイベントについて、カーネルからメッセージを受け取る。

2. **制御ポート** (control port):各メモリオブジェクトに対して、カーネルに

よって制御ポートが作成される。カーネルはこれを使って、オブジェクトに関係するメッセージをメモリ管理者から受け取る。

3. **名前ポート** (name port):各メモリオブジェクトに対して、カーネルは名前ポートも作成する。これはオブジェクトを特定するための一種の名前として使われる。名前ポートはメッセージを受け取るためには使われない。しかし、参照ポイントとしては使われる。例えば、アドレス空間の各リージョンはオブジェクトとして扱うことができる。この場合、カーネルは各リージョンに対して名前ポートを持つ。そこで、スレッドはアドレスをカーネルに与えて、そのアドレスが属しているリージョンを問う。この時、カーネルは答えとしてリージョンの名前ポートを返す。同じリージョンに属しているすべてのアドレスは同じ名前ポートによって特定されることに注意してほしい。

ここで、メモリ管理者とカーネルがどのようにメモリオブジェクトを管理し、メモリオブジェクト中のデータに対するユーザのアクセス要求を満たすかについてみることにしよう。Machが提供しているvm_map呼び出し(仮想アドレス空間を操作するための他の呼び出しとともに提供されている)を使って、メモリオブジェクトを呼び出しプロセスの仮想アドレス空間にマッピングする。スレッドはメモリオブジェクトのデータにアクセスする必要があるとき、vm_map呼び出しを実行する。メモリオブジェクトに対する最初のvm_map呼び出しであった場合、カーネルはメモリオブジェクトに対する制御ポートと名前ポートを作成し、vm_map呼び出しに含まれていたオブジェクトポートに対してmemory_object_initメッセージを送る。そのメモリオブジェクトを管理するメモリオブジェクト管理者は、そのオブジェクトに関するサポートの一部として、このオブジェクトポートを提供する。応答として、メモリ管理者はmemory_object_set_attributesメッセージをカーネルに送り、オブジェクトの属性について通知する。最初は、オブジェクトのすべてのページが読み取り不可能／書き込み不可能と記されている。したがって、オブジェクトに対する最初のアクセスはトラップを生じる。vm_map呼び出しを実行したスレッドはそこでブロックが解けて、実行が許される。

スレッドがメモリオブジェクトのデータを読み取り／書き込みしようとすると、ページフォールトがおき、そして、カーネルはメモリオブジェクトのオブジェクトポートを通じてmemory_object_data_requestメッセージをメモリ管理者へ送る。このメッセージを受けると、メモリ管理者は関係のあるオブジェクトに対して適切な何らかのメソッドを行うことでそのページをフェッチし、

memory_object_data_provided メッセージでそのページへのポインタを返すか、あるいはカーネルに適切な誤りコードを返す。そして、カーネルはそのページをフォールトしているスレッドのアドレス空間にマッピングし、そのスレッドのブロックをといて実行を続行できるようにする。

ユーザが記述した特別なメモリ管理者に加えて、Mach システムは**デフォルトメモリ管理者** (default memory manager) を提供している。このメモリ管理者は幾つかの理由で必要である。例えば、メモリ管理者が割り当てられていないアドレス空間のリージョンを管理するためや、オブジェクトの特別なメモリ管理者がバグによってクラッシュしたときにデフォルトの方法でメモリオブジェクトを管理するために必要である。デフォルトメモリ管理者はユーザが記述したメモリ管理者と同一のインタフェースを持っている。しかし、デフォルトメモリ管理者はディスクに書く必要のあるデータを格納するのに標準のファイルシステムを使う。スワップ領域を使う UNIX とは異なり、デフォルトメモリ管理者はスワップ空間として一時的なファイルを使う。プロセスが仮想アドレス空間のリージョンを割り当てるために vm_allocate 呼び出しを実行した場合、実際には、デフォルトメモリ管理者はその呼び出しの応答として 0 で埋めたページを提供し、これによって管理されてるオブジェクトをマッピングすることに注意してほしい。

常に十分な空きページフレームがあることを保証するために、Mach はカーネルの一部としてページアウトデーモンを提供している。これはカーネルのスレッドとして動作しており、ページ置き換えの犠牲を選択するために先入れ先出し (First In First Out : FIFO) のアルゴリズムを使用する。ページアウトデーモンは時々起動して、空きページフレームの数を検査する。もし十分な空きページフレームがない場合、置き換えの対象として 1 つ又は複数のページを選択する。置き換えの対象として選ばれたページがダーティであった場合、それは momory_object_data_write メッセージの責任によってメモリ管理者に送られる。このメッセージを受け取ると、メモリ管理者はページをディスクに書き、それが終了するとカーネルに通知を行う。

メモリ共有

Mach においてスレッドを使用すると、プロセスのスレッドは同じアドレス空間を自動的に共有することができる。したがって、1 つのプロセスのスレッド間でメモリを共有するための特別な機構は必要ない。しかし、2 つ又は複数のプロセスでメモリを共有するために、Mach は次に説明する 3 つの機構を提

供している。

書き込み時コピーによる共有

これまでに、カーネルを含むMachの各プロセスがカーネルによって管理される自分自身のページ化されたアドレス空間を持っていることを示してきた。したがって、同じノードの2つのプロセスは、両方のプロセスのページテーブルにメモリリージョンのエントリを持つことで、メモリリージョンを共有することができる。この技法は書き込み時コピーのメモリ共有の機構において使われている。この機構では、2つ又は複数のプロセスが共有するメモリページに対するエントリをそれぞれのプロセスのページテーブルに持ち、それらを読み取り専用にすることで共有化を行う。プロセスのうちの1つが共有されたページの自分自身の論理的なコピーに対して書き込みを行うとするまで、この共有は続く。つまり、書き込みによって、ページの実際のコピーがそのプロセスのために作られ、そのプロセスのページテーブルには書き込み可能と記された自分自身の版のページが存在するようになる。そのページを共有していた他のプロセスは読み取り専用のモードでそのページの古い版を使いつづける。

アドレス空間の継承の制御

この機構によって、プロセスを作成するときに、選択的なメモリ共有が可能となる。UNIXと同様に、Machで新しいプロセスが作成されると、基本的には既存のプロセスのコピーとして作成される。しかし、Machのプロセス作成の機構はUNIXのものとは次の点で異なる。

1. UNIXでは、子プロセスは常にforkシステム呼び出しを実行したプロセスのクローンである。しかし、Machでは、子プロセスはプロトタイプと呼ばれる異なるプロセスのクローンになることができる。

2. UNIXでは、子プロセスは親プロセスのアドレス空間全体を継承する。しかし、Machでは、アドレス空間の継承を制御できる。制御されたアドレス空間の継承に対して、Machでは、プロセスがそのアドレス空間の各リージョンに対して次の3つの継承属性の1つの割り当てることができる。

(a) アクセスなし。この属性を持つリージョンは子プロセスによって継承されない。すなわち、リージョンは子プロセスのアドレス空間の一部にはならない。

(b) 共有アクセス。この属性を持つリージョンはプロトタイププロセスと子

プロセスとの間で共有される。すなわち、そのリージョンのページはプロトタイププロセスと子プロセスの両方のアドレス空間に存在する。どちらかのプロセスによる変更はもう一方のプロセスにも可視である。

(c) コピーアクセス。この属性を持つリージョンは子のアドレス空間にコピーされマッピングされる。このオプションによって UNIX の fork システム呼び出しを模擬することができる。しかし、Mach では、子プロセスが作成されたとき、このリージョンはコピーされない。効率を目的として書き込み時コピーの機構が使われている。

分散共有メモリ

Mach は共有メモリを持たない別々のマシンにあるプロセスでメモリを共有するための直接的な機構を提供していない。しかし、Mach の外部メモリ管理者の概念によって、ユーザが自分自身の分散共有メモリシステムを実装することができる。例えば、ページ準拠の分散共有メモリシステムを実装するために、新しいメモリオブジェクト、共有ページを定義することができる。これらのオブジェクトを管理するために、1 つ又は複数の特別なメモリ管理者を実装することができる。メモリ管理者は必要に応じて単純にあるいは複雑にすることができ、それに依存して、第 5 章で説明したもののうち、一貫性プロトコルやデータ位置特定機構が使われる。

12.4.5　プロセス間通信

基本抽象概念

Mach のプロセス間通信で使われている 2 つの基本抽象概念はポートとメッセージである。Mach のポート (port) は一方向の通信チャンネルである。それは、論理的に上限のあるメッセージキューである。メッセージ (message) はデータオブジェクトのタイプ付きの集合である。別のプロセスと通信するためには、プロセスは受信プロセスに付随しているポートにメッセージを送る。メッセージは受信側がポートからそれを受け取るまでポートにキューイングされる。送信側と受信側はそのポートにアクセスするために許可を持っていなければならない。この許可はケーパビリティの形式をとっている。例えば、送信側はポートに対する send 権のあるケーパビリティを持たなければならないし、受信側はそのポートからメッセージの受信を行うための receive 権のあるケーパビリティが必要である。システムがメッセージの配送を送った順序で保証す

るという点で、ポートは、信頼性のある、順序付けられたメッセージストリームをサポートしている。

ポートの管理

ポートはカーネルによって管理され保護されている。ポートはスレッド単位ではなくプロセス単位に基づいて追跡されている。したがって、プロセスのスレッドによって作成されたすべてのポートがそのプロセス内で一意である。カーネルはどのスレッドがどのポートを作成したかについては記録を持っていない。

保護はポートに関する情報をカーネルの中に安全に持っていることで保証されている。ここではユーザプロセスがポートを修正することはできない。更に、プロセスのスレッドは、そのプロセスが適切なポートケーパビリティを所有しているときのみに、ポートに対してメッセージを送信あるいは受信することができる。ポートケーパビリティは、ポート識別子 (ポートへのポインタ) とケーパビリティの持ち主がそのポートに対してどのアクセスを持っているかを示す権利の欄から構成される。ポートに対して定義することのできる三種類の権利タイプは、send、send_once、receive である。send 権利を持つケーパビリティによって、その所有者は指定されたポートに何度でもメッセージを送ることができる。send_once 権利によって、ケーパビリティの所有者はそのポートに 1 つのメッセージだけを送信することができ、送信後はカーネルがそのケーパビリティを破棄する (この機構は要求応答プロトコルに便利である)。receive 権によって、ケーパビリティの所有者は、そのポートからメッセージを読むことができる。任意の時点で、ポートに対する receive 権を持つプロセスは 1 つに制限されるが、複数のプロセスが同時に send 権を持つことは可能である。receive 権のあるポートケーパビリティを持っているプロセスはそのケーパビリティを別のプロセスに対してメッセージを使って送ることができる。しかし、そうすることによって、これを送信した側は receive 権を失い、再び、それを受信した側が receive 権を獲得する。各プロセスに対して、カーネルはケーパビリティのリストを管理している。このリストには、プロセスが異なるポートに対して所有している権利の種類に関する完全な情報がある。ポートはプロセス単位を基本として追跡されるので、プロセス中のすべてのスレッドはプロセスのポートケーパビリティの同じ所有者として考えられる。

receive 権のあるポートケーパビリティを所有しているプロセスが終了するか殺されるかすると、ポートはもはや使うことができず、したがって、たとえ配送していないメッセージがあろうともカーネルによって破棄される。そして、

カーネルはポートの send ケーパビリティを検索して、それらを死んだと記す。カーネルによって死んだと記されたケーパビリティによってメッセージを送信しようとしても、これは失敗する。カーネルは送信側に対して適切な誤りコードを返す。一方、ポートに対する send 権のあるケーパビリティを持つプロセスがないとき、カーネルは (オプションではあるが)receive 権のケーパビリティを持っているプロセスにメッセージを送り、残っている送信側がいないことを通知する。

ポートに関連したメッセージキューは、有限の長さであり、一杯になるかもしれない。満杯のキューに対してメッセージを転送することに関する問題を処理するために、幾つかのオプションが提供されている。例えば、キューが一杯になったとき、送信側は送信をアボートし、キューのスロットが利用可能になるまでブロックすることができる。あるいは、カーネルに配送をしてもらうこともできる。後者では、カーネルは 1 つのメッセージキューとして働き、既にキューイングされているメッセージを配送するまでは、送信側からのどのメッセージも受け付けることはできない。

Mach は多くのポートをグループ化して**ポート集合** (port set) にする機能を提供している。ポートは同時に最大 1 つのポート集合に属することができる。ポート集合からはメッセージを受信することだけができる。つまり、ポート集合にメッセージを送ることはできない。更に、ポート集合に属しているポートはメッセージを受信するために直接使うことはできない。なぜなら、ポート集合に属するポートへ送られるすべてのメッセージは共通のキュー、ポート集合キューにキューイングされるからである。ポート集合に対して受信が実行されるとき、カーネルはポート集合キューから 1 つのメッセージを返す。キューが空の場合、上記に説明したオプションの 1 つを選択しこの問題に対処することができる。ポート集合を作成する機能は、複数のポートで送られてくる要求を処理するサーバプロセスを記述するのに特に便利である。例えば、複数のオブジェクトをサポートするサーバなどがそれである。そのようなサーバはサポートしている多くのオブジェクトのそれぞれに異なるポートを管理することができ、それぞれに専用のスレッドを持つ必要なくメッセージを受け取ることができる。

メッセージパッシング

基本的なプロセス間通信には、送信プロセスがポートにメッセージを送ることと、受信プロセスがその同じポートからメッセージを受信するということが

含まれる。メッセージは同期あるいは非同期に送信及び受信することができる。RPC 機構もまた実装されており、そこでは送信側はメッセージを送った後は応答が返ってくるまでブロックする。

メッセージは固定長のヘッダと可変個の型付きのデータから構成される。ヘッダには、宛先ポートのケーパビリティ名、返りのメッセージを送るべき応答ポートのケーパビリティ名、メッセージサイズ、同期、非同期、あるいはRPC タイプの通信のどれが望まれているか、また、send と receive が成功しなかった場合はどうするか、ポートのメッセージキューが一杯だった場合はどうするか、などの情報が含まれる。

メッセージのデータ部分は、可変個の型付きデータオブジェクトから構成される。各データオブジェクトは、**メッセージ内データ** (in-line data) として、あるいは、**メッセージ外データ** (out-of-line data) として表現することができる。前者の表現では、データオブジェクトはメッセージの本体に含まれる。一方、後者の表現では、メッセージ本体はデータオブジェクトへのポインタを含んでいる。メッセージ内の各データオブジェクトに付随したものとして記述子があり、これにはデータオブジェクトの型とそのサイズなどの情報が含まれている。この情報は受信側がデータを正確にアンパックするために必要である。それは、送信元と宛先のマシンで異なる内部表現を持っているマシン間で会話を行うのに便利である。プロセスがケーパビリティを別のプロセスに送りたい場合、メッセージのデータ欄には、ポートケーパビリティが含まれる。

メッセージの本体にメッセージ外データによる表現を使う機能は、1つのメッセージでプロセスのアドレス空間全体を転送する手段を提供する。更に、送信プロセスと受信プロセスの両方が同じノードにある場合、Mach は書き込み時コピーの機構を使って、送信側から受信側へメッセージ外データを転送する。すなわち、送信側のアドレス空間から受信側のアドレス空間に物理的にデータをコピーする代わりに、カーネルは単にデータオブジェクトのすべてのページに関する情報を受信側のページテーブルに置いてこのテーブルを更新する。そして、書き込み時コピーとする。データオブジェクトの記述子のビットによって、データオブジェクトに対応するリージョンは、送信側のアドレス空間から削除されるか (送信側のページテーブルから対応するエントリを削除する)、そこに残される。後者では、データのページは受信側がそれに書き込みを行おうとすると、受信側のアドレス空間にコピーされる。そして、受信側のページテーブルの対応するエントリが適切に更新される。これは、どの修正も送信側が使っているオリジナルの版のデータに影響を与えないことを保証するために

行われる。

書き込み時コピーの機構によって、メッセージパッシングは非常に効率的になっている。なぜなら、多くの場合データのコピーは必要とされないからである。本質的に、メッセージパッシングは仮想メモリ管理の機構を使って実装されている。

ネットワークワイドな IPC

ネットワークワイドな IPC に対して、Mach は**ネットワークメッセージサーバ** (network message server) と呼ばれるユーザレベルのサーバと、システムの他のノードからメッセージを受信するポートである**ネットワークポート** (network port) を使用している。システムの各ノードにはネットワークメッセージサーバが存在する。すべてのネットワークメッセージサーバは協調して動作し、ノード間のメッセージを透過的な方法で処理する。ネットワークメッセージサーバによって行われる 3 つの主な仕事は次の通りである。

1. ネットワークワイドな IPC 機構をネットワーク透過にする。

2. 送信側のコンピュータのフォーマットから受信側のコンピュータのフォーマットにメッセージデータを変換することで異機種をサポートする。メッセージの各データオブジェクトに対する記述子にある情報がこの目的のために使われる。

3. 他のネットワークメッセージサーバを認証することで、メッセージデータが認証されていないユーザの手に落ちることを防ぐ。

ノードのプロセスは、ローカルなネットワークメッセージサーバがローカルなプロセスからメッセージを受信するために作成したポートに対する send 権を自動的に継承する。ノードのネットワークメッセージサーバにより、ローカルプロセスはネットワークポートをそれに登録することができ、すべてのノードのネットワークメッセージサーバが互いに通信を行い、システムのすべてのネットワークポートに関する分散したデータベースを管理することができる。プロセスはローカルなネットワークメッセージサーバに対して、データベースにある名前を検索するように依頼したり、あるいは、メッセージにあるポートケーパビリティを受信することで、ネットワークポートに対する send 権をプ

ロセスは獲得することができる。

ノード A のプロセス P_1 からノード B のプロセス P_2 へメッセージを送る基本的な方法を図 12.10 に示す。これは次のステップから成る。

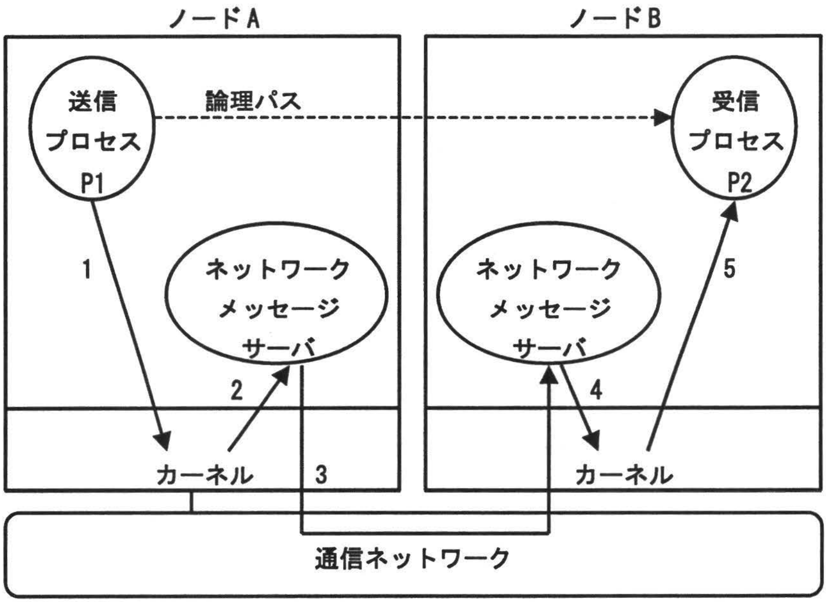


図 **12.10** Mach におけるネットワークワイドな IPC 機能

1. プロセス P_1 は送信するメッセージのヘッダと本体を準備し、ローカルな IPC に場合と同様に送信メッセージに対して message_send システム呼び出しを実行する。

2. そのシステム呼び出しによってカーネルに対するトラップが起こる。カーネルはメッセージヘッダから受信側のポート番号を抽出し、このポートがローカルではないことを確認し、そのメッセージをローカルなネットワークメッセージサーバへフォワードする。

3. 次にノード A のネットワークメッセージサーバはネットワークポートのデータベースを調べ、どのノードにメッセージをフォワードすべきかを見つけ

出す。そして、プロセス P_1 のメッセージを含むネットワークメッセージを作り、ネットワークを越えてノード B のネットワークメッセージサーバへそれを送信する。2 つのネットワークメッセージサーバ間のメッセージがセキュリティを目的として暗号化される場合もある。

4. ノード B のネットワークメッセージサーバはメッセージ中のネットワークポート番号を見つけ出し、管理しているマッピングテーブルを使ってその番号を同等なローカルなポート番号にマッピングする。必要があれば、メッセージデータをノード A のコンピュータ表現から自分自身のコンピュータ表現に変換する。そして、このメッセージを摘出されたローカルポートへ送るために message_send システム呼び出しを実行する。

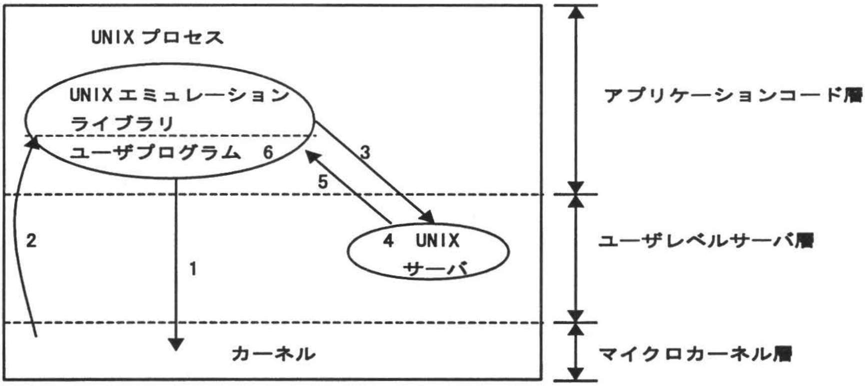
5. そのシステム呼び出しによってカーネルに対するトラップが起きる。カーネルは受信側のポート番号をメッセージヘッダから取り出し、このポートがプロセス P_2 が receive 権を持っているローカルなものあることを確認し、P_2 が message_receive 呼び出しを実行したとき、そのメッセージをプロセス P_2 に提供する。このように、メッセージは送信側から受信側へ透過的な方法で転送される。

ネットワークメッセージサーバはユーザレベルのプロセスであるので、ユーザが設計し、データタイプ表現、ネットワークで使われるセキュリティの程度やタイプ 、そして接続されたネットワークの種類に特化したプロトコルの使用を柔軟に選択できるすることが可能である。しかし、この柔軟性は性能を犠牲にして達成される。なぜなら、純粋なカーネルによる実装 (これは多くの他の分散システムが使っている) は、ノード間のメッセージをユーザレベルのサーバにフォワードしてもらうアプローチよりはより優れた性能を有するからである。

12.4.6　UNIX エミュレーション

Mach は 4.3 BSD UNIX をエミュレートしている。Mach における UNIX エミュレーションに使われている基本的なアプローチを次に簡単に説明する。詳細については [Golub et al. 1990] に見ることができる。

図 12.11 に示すように、Mach において UNIX エミュレーションを行うのに使われている 2 つのソフトウェア構成要素は、**UNIX エミュレーションライブラ**



1 カーネルにトラップをかける
2 UNIX エミュレーションライブラリに制御が移る
3 UNIX サーバに対して RPC を実行する
4 システム呼び出しが実行される
5 応答が返される
6 UNIX プロセスの一部であるユーザプログラムに制御が戻される

図 **12.11** Mach における UNIX エミュレーション

リ (UNIX emulation library) と **UNIX サーバ** (UNIX server) である。UNIX エミュレーションライブラリは、UNIX プロセスをエミューレートしているそれぞれのプロセスに個別のリージョンとしてリンクされる。このリージョンは UNIX プロセスが fork するとき、すべての UNIX プロセスによって/etc/init から継承される。exec システム呼び出しは、エミュレーションライブラリを置き換えるのではなく、プロセスのアドレス空間のユーザプログラムの部分を置き換えるように、変更が施されている。UNIX サーバには、大きな UNIX コードが含まれており、ここには UNIX システム呼び出しに対応したルーチンが存在する。それは C スレッドの集合として実装されている。

トランポリン機構 (trampoline mechanism)[Tanenbaum 1995] がエミュレーションライブラリのコードを起動するために使われている。この機構では、エミュレーションライブラリのコードは直接的にユーザアプリケーションからは起動されない。システム呼び出しによってカーネルに対してトラップがおきたとき、エミュレーションライブラリはカーネルから制御をもらう。このため、Mach は task_set_emulation という名前の呼び出しを提供している。これは、エミュレーションライブラリの中のハンドラのアドレスを、与えられたシステ

ム呼び出し番号に設定する。システムの初期化の時点で、各 UNIX システム呼び出しに対してこれが呼び出される。これによって、すべてのシステム呼び出しがカーネルによって管理されているシステム呼び出しリダイレクションテーブルに登録される。

この設定によって、UNIX プロセスがシステム呼び出しを実行し、カーネルへトラップをあげると、カーネルはシステム呼び出しリダイレクションテーブルを使って、すぐに制御を同じプロセスのエミュレーションライブラリへ移す。エミュレーションライブラリが制御をもらうと、どのシステム呼び出しが起動されたかを判断するためにマシンのレジスタをテストする (この時、すべてのマシンレジスタはトラップ の時点と同じ値を持っていることに注意してほしい)。そして、エミュレーションライブラリはシステム呼び出し番号と必要なパラメータをメッセージにパックして、UNIX サーバに対して RPC を実行する。RPC メッセージを受けると、UNIX サーバはシステム呼び出し番号とパラメータをそこから取り出し、システム呼び出しを実行する。そして、応答をエミュレーションライブラリに返す。応答を受け取ると、エミュレーションライブラリはカーネルを通さず直接に制御をユーザプログラムへと渡す。全体の過程を図 12.11 に要約する。

12.4.7　Mach インタフェースジェネレータ

プログラマはシステム呼び出しインタフェースを直接使って Mach システムのアプリケーションを書くことができる (Mach の幾つかのシステム呼び出しは先の章で説明した)。しかし、システム呼び出しレベルでの作業は多くのプログラマにとって退屈であり繰り返しの多いものである。例えば、サーバプロセスからサービスを受けるクライアントプログラムは、メッセージを作成しサーバへ送り、そして応答を受信するのを待つプログラムを書かなければならない。同様に、クライアントにサービスを提供するサーバプログラムは、メッセージを受け付け、それをアンパックし、適切なルーチンにディスパッチし、そのルーチンがデータを処理し終わったら、応答を行うプログラムを書かなければならない。すべてのサービスに対してスタブ手続き (繰り返しの多いコードの大部分を生成する手続き) を提供し、それを透過的な共有ライブラリに置くことで、プログラマの作業は大きく単純化される。このアプローチによって、アプリケーションプログラマは単に手続き呼び出しを実行するだけでサービスを利用することができる。アプリケーションプログラマはシステム呼び出しやメッ

セージを送受信するためのコードを記述する必要はない。これを機能させるために、Machはインタフェース仕様言語とMach Interface Generator(MIG)と呼ばれるコンパイラを提供している。

MIG言語によって、協調して計算を行うエンティティ間のインタフェースを指定することができ、特定の言語やマシンアーキテクチャに依存することなくインタフェースを管理することができる。一方、MIGコンパイラはMach環境でサポートされる各プログラミング言語に対して、仕様からインタフェースのコードへ変換を行う。Mach環境の言語には、C、COMMON LISP、Ada、そしてPascalなどがある。コンパイラによって生成されたインタフェースコードには、通信、型検査のための実行時サポート、型変換、同期化、例外処理のコードがある。

幾年かを経て、MIGは価値あるツールであることが証明されてきた。それは主に次の利点による。

1. プログラマを、メッセージデータフォーマット、オペレーティングシステムの特性、特定の同期化の詳細に関する注意から解放することで、分散アプリケーションのプログラミング作業を簡単にする。

2. 異なる言語によるプログラマの作業の協力を改善する。これはクライアントとサーバの両方はMach環境でサポートされる任意の言語で記述できるからである。MIGコンパイラは自動的に、言語文法、レコード欄レイアウト、手続き呼び出しセマンティクス、例外処理セマンティクスの違いを担当してくれる。

3. プロセス間の統一的なメッセージレベルのインタフェースを提供することで、システムの標準化を促進している。

4. プログラムインタフェースが変更されたときは、複数言語によるインタフェースの再プログラミングのコストを減らす。

12.5 CHORUS

Chorusはマイクロカーネル準拠の分散オペレーティングシステムであり、1979年にフランスの政府系研究所であるINRIA(Institute National de Recherche en Informatique et Automatique)の研究プロジェクトとして始まっ

た。今日までに、Chorus は 4 つの主要な版 (第 0 版から第 3 版) を経てきた。第 0 版 (1979 - 1982) はアクタ (actor) と呼ばれる通信プロセスの集合体として分散アプリケーションをモデル化するために設計された。第 1 版 (1982 - 1984) は第 0 版の設計を共有メモリマルチプロセッサシステムから分散メモリマルチプロセッサシステムへ移植することが目的であった。それは、構造化されたメッセージとフォールトトレラント性に対する幾つかのサポートによる追加的な特徴を持っている。第 2 版の主な目標は UNIX ソースコードとの互換性の特徴をシステムに加えることであった。これによって既存の UNIX プログラムが再コンパイルした後 Chorus で動作するようになった。1986 年、Chorus チームは INRIA を離れ、Chorus Systems という名前の新しい会社を形成した。そして、Chorus は製品となった。彼らは 1987 年に第 3 版を始めた。主な目的は研究システムから製品へと変えることであった。このために、最初の目的は UNIX とのバイナリ互換性を提供し、再コンパイルなしに UNIX プログラムを Chorus 上で動作できるようにすることであった。他の分散オペレーティングシステムに見られる鍵となる概念が第 3 版には含まれている。特に、メッセージ準拠の通信機構は V-System から借りてきた。高速なプロセス間通信、分散仮想メモリ、外部ページャの概念などは Mach から借りてきた。グローバルな名前付けと保護に対してケーパビリティを使うアイディアは Amoeba から借りてきた。第 3 版は RPC 機能、リアルタイム操作のサポート、マルチスレッドの特徴も持っている。Intel 80x86 ファミリ、Motorola 68000 と 88000 ファミリ、Inmos Transputer などの広範なハードウェアで商用の製品が利用できる。

第 3 版の詳細を以下で述べる。Chorus に関する次の説明は [Pountain 1994, Armand et el. 1986, Rozier et al. 1988, Guillemont 1982, Abrossimov et al. 1992, Batlivala et al 1992, Lea et al. 1991, Lea et al 1993, Tanenbaum 1995, Coulouris et al. 1994] をもとにしている。

12.5.1　設計の目標と主な特徴

Chorus の設計は次の研究と設計目標に影響を受けている。

UNIX エミュレーションと拡張

Chorus の主な目標の 1 つは UNIX 互換性を提供することであった。これによって、既存の UNIX プログラムが Chorus 上で動作することができる。これは初期の目標ではなかったが、後になって商業的な成功のためには重要である

ため実現された。したがって、Chorus 第2版は UNIX ソースコード互換性を提供するように設計されている。この目標を達成するために、Chorus のもともとのカーネルは再設計され、できるだけ多くの機能をカーネルからユーザアドレス空間へと移動することで、マイクロカーネルへと変換された。そして、幾つかのプロセスが UNIX エミュレーションを行うために、ユーザアドレス空間に加えられた。後に、第3版では、Chorus/MiX(Mix は Modular UNIX を表す) と呼ばれる UNIX エミュレーションサブシステムが Chorus マイクロカーネルの上に構築され、UNIX System V とのバイナリ互換性を提供している。第2版のマイクロカーネルは、更に洗練され、ソースコードによる UNIX エミュレーションを行うために加えられた部分は移動され、新しい UNIX エミュレーションサブシステムの中に置かれた。**4.3** BSD UNIX エミュレーションもまた現在実装されているところである。

UNIX エミュレーションに加えて、Chorus の設計は UNIX の拡張を提供している。これにより、UNIX エミュレーションのユーザが UNIX プロセスから Chorus が提供している拡張機能を使うことができる。1つのプロセス中でマルチスレッドを利用する機能と、遠隔のノードに新しいプロセスを作成する能力である。

オープンシステムアーキテクチャ

Chorus のもう1つの重要な特徴は、マイクロカーネルのサポートである。これは、新しいオペレーティングシステムを構築するための基盤とモジュール性のある方法で既存のオペレーティングシステムをエミュレーションするための基盤を提供している。この特徴によって、UNIX System V、BSD UNIX、OS/2、MS-DOS などの複数のオペレーティングシステムインタフェースが同時に同一のマシンに存在することができる。したがって、異なるマシンで現在動作している既存の幾つかのアプリケーションを同一のマシンで実行することが可能であろう。

効率的かつ柔軟な通信

Chorus において使われている基本的な通信パラダイムはメッセージパッシングである。メッセージパッシングは共有メモリにくらべて非効率であるという悪評があるので、Chorus の設計者は IPC システムを最適化するために多大の努力を行った。IPC システムは異なるタイプの通信を処理する際に高い次元の柔軟性を提供している。この IPC システムは次の特徴を有している。

1. 非同期メッセージパッシングと要求/応答タイプのインタラクションの両方を提供している。

2. 最大一回のセマンティクスを提供する RPC 機能がある。また、2 つのカーネルプロセス間で通信を行うための軽量 RPC 機能もある。

3. 柔軟なグループ通信機能があり、メッセージをすべてのメンバに送信すべきか、いずれか 1 つのメンバに送信すべき、特定のメンバに送信すべきか、指定されたノードにはないいずれかのメンバに送信すべきか選択できる。

4. Amoeba において使われていたのと同様にケーパビリティ準拠のアクセス制御機構を用いて、安全なメッセージ通信を保証する。

5. メッセージの送信側と受信側の両方が同一のマシンにあるとき、コピーすることなく大量のデータを転送するのに、Mach の書き込み時コピーの機構を使う。

透過性

Chorus は 2 つのタイプの透過性を提供している。つまり、ネットワーク透過性とサービス再構成透過性である。ネットワーク透過性は、1 つのグローバル名前空間と**ネットワーク管理者** (network manager) と呼ばれるユーザレベルのサーバを使うことで実装されている。ネットワーク管理者は Mach のネットワークメッセージサーバに似たものである。一方、サービス再構成透過性は、これによって、サービスを利用しているユーザに通知することなく動的にサービスを再構成できる。この透過性は、ポートグループとポート移動をサポートすることで実現されている。Chorus は完全にネットワーク透過ではないことに注意してほしい。なぜなら、幾つかのシステム呼び出しはローカルな場合にのみ動作するからである。

柔軟なメモリ管理

柔軟なメモリ管理に関して Chorus が提供する 2 つの主な機能は、複数のユーザレベルのメモリ管理者のサポートと、ページ分散共有メモリのサポートである。前者の機能は、Mach 形式の外部ページャによって実装されており、それは Chorus ではマッパ (mappers) と呼ばれている。

リアルタイムアプリケーションのサポート

Chorus のもう 1 つの重要な目標は、リアルタイムアプリケーションのサポー

トである。この目標を達成するために、Chorusはスレッド優先度の柔軟な割り当てを提供し、カスタマイズしたスレッドスケジューリング方針の実現が可能である。更に、リアルタイムプログラムは部分的にはカーネルモードで実行することができ、あいだにソフトウェアを挟むことなく、直接マイクロカーネルにアクセスすることができる。

オブジェクト指向プログラミングインタフェース

Chorusの比較的新しい目標は、細粒度のオブジェクト指向言語とアプリケーションに対してシステムレベルのサポートを提供することであり、新しいオブジェクト指向のプログラムと古いUNIXプログラム干渉することなく同じマシンで動作できるように、これらを提供することである。この目標を達成するために、Chorusはマイクロカーネルの上にChorus Object-Oriented Layer(COOL)と呼ばれるサブシステムを設計した。

12.5.2 システムアーキテクチャ

図12.12に示すように、Chorusのシステムアーキテクチャは主に次の階層から構成されている。

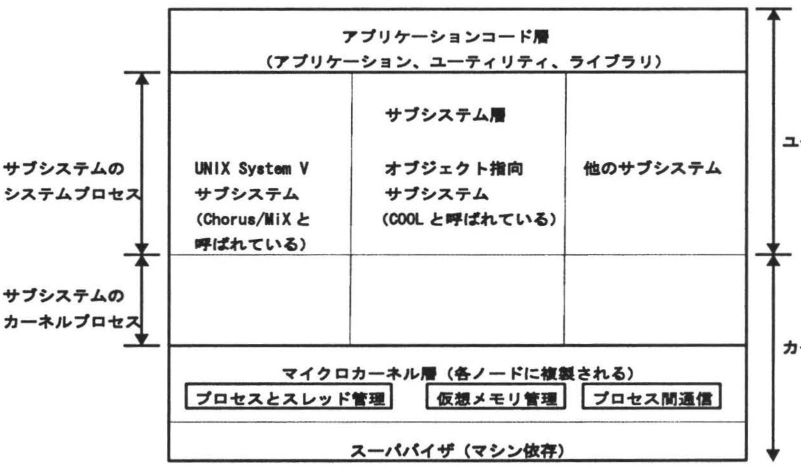


図 12.12 Chorusシステムアーキテクチャ

1. **マイクロカーネル層**:マイクロカーネル層は最も低い層であり、Chorus では**核** (nucleus) と呼ばれている。この層はシステムの各ノードにある。それは、4 つの構成要素から成り、そのうち 3 つはマシン非依存であり、残りの 1 つがマシン依存である。マシン依存の部分は**スーパバイザ** (supervisor) と呼ばれ、ハードウェアを直接管理したり、トラップ、例外、割り込みなどを捕まえる。また、コンテキスト切り替りも処理している。この部分は Chorus が移植される新しいハードウェアのそれぞれに対して書き直す必要がある。3 つのマシン非依存の部分については、1 つはプロセスとスレッド管理を行うためのものであり、もう 1 つは仮想メモリ管理のためのものであり、更に最後の 1 つはプロセス間通信を処理するためのものである。カーネルの 4 つの構成要素はモジュール性の高い方法で構成されており、どれか 1 つを別のものに影響を与えることなく変更することができる。

2. **サブシステム層**:マイクロカーネル層は、モジュール性のある方法で、新しいオペレーティングシステムを構築したり、既存のオペレーティングシステムをエミュレートしたりするための基礎を提供する。そのような方法で新しく構築されたシステムやエミュレートされたシステムは**サブシステム** (subsystem) と呼ばれている。このようなすべてのサブシステムによってサブシステム層が形成され、その層はマイクロカーネル層の上に位置している。サブシステムはきちんと定義されたインタフェースをユーザに見せる。例えば、そのようなサブシステムの 1 つが MiX と呼ばれる UNIX System-V エミュレータである。これは UNIX インタフェースを提供しており、これによって UNIX プログラムが Chorus 上で動作することができる。もう 1 つのそのようなサブシステムは、COOL と呼ばれるオブジェクト指向のサブシステムである。COOL は細粒度のオブジェクト指向言語とアプリケーションに対するシステムレベルのサポートを提供する。

各サブシステムは Chorus プロセスの集合である。これらのプロセスには 2 つのタイプがある。1 つは**カーネルプロセス** (kernel process) であり、もう 1 つは**システムプロセス** (system process) である。カーネルプロセスはカーネルモードで実行し、システムプロセスはユーザモードで実行する。カーネルプロセスは別のカーネルプロセスを呼び出すことができ、マイクロカーネルを起動してサービスを得ることができる。一方、システムプロセスは互いにメッセージを送ったり、カーネルプロセスにメッセージを送ることができ、マイクロカーネルに対する呼び出しを実行することもできる。

サブシステム層においてカーネルプロセスを使う背景にある基本的なアイディアは、マイクロカーネルのサイズをに大きくしたり、複雑にしたりすることなしに、マイクロカーネルの機能を拡張する方法を提供することであった。これは、システムの実行中に、カーネルプロセスを動的にロードしたり、削除する柔軟性を提供することで達成されている。この機能によって、動的にシステムを構成することができ、マイクロカーネルを再コンパイルや再リンクすることなしに、システムの特定のノードのハードウェア構成要素に合うように変更することができる。例えば、ディスクサーバはカーネルプロセスとして実装されており、ディスクレスのワークステーション上にはロードする必要がない。

カーネルプロセスはマイクロカーネルとアドレス空間を共有している。したがって、カーネルプロセスはそれがロードされた後は再配置しなければならない。

3. アプリケーションコード層:アプリケーションコード層はサブシステム層の上位にある層である。この層には、アプリケーション、ユーティリティ、そしてライブラリなどのユーザアプリケーションがある。この層のユーザプロセスとサブシステム層にあるシステムプロセスはユーザアドレス空間を共有する。

ユーザプロセス (user process) はマイクロカーネルを直接呼び出すことはできない。使用しているサブシステムが提供するシステム呼び出しを実行することのみができる。これを保証するために、マイクロカーネルはどのユーザプロセスがどのサブシステムを使用しているのかを追跡しており、ユーザプロセスが別のサブシステムが提供するシステム呼び出しを実行することを許さない。しかし、リアルタイムプロセスには特権があり、ユーザプロセスではなくシステムプロセスとして実行することができる。これによって、リアルタイムプロセスは何らソフトウェアを介することなく、マイクロカーネルに直接アクセスすることができる。

12.5.3 鍵となる抽象概念

Chorus の設計で使われている鍵となる抽象概念を次に示す。

1. アクタ (actor):Chorus における“アクタ”は本書で説明されている“プロセス”の概念に対応するものである。アクタは 1 つ又は複数のスレッドに対して実行環境を提供し、アドレス空間と、メッセージを送受信するために使う

ポートの集合を持っている。混乱を避けるために、以降の説明では、“プロセス”という言葉を“アクタ”の代わりに使うことにする。ただし、Chorus システム呼び出しにおいては例外である。

2. **スレッド** (thread):Chorus の“スレッド”は本書における“スレッド”の概念と同じである。Chorus では、スレッドはカーネルによって管理されている。すなわち、スレッドの作成、破棄、スケジューリングはカーネルによって行われ、カーネルのデータ構造の更新を含んでいる。

3. **リージョン** (region):各プロセスには 1 つ又は複数のリージョンから構成されるアドレス空間がある。リージョンは連続した仮想アドレスの領域であり、プログラムやファイルなどの何らかのデータに関連づけられている。プロセスのリージョンはオーバラップすることはない。リージョンによって占められているアドレス空間の部分だけが、所有プロセスのスレッドからアクセス可能である。ページ仮想メモリシステムでは、リージョンはページ境界にそって配置されており、リージョンは 1 つ又は複数のページから構成される。

4. **セグメント** (segment):セグメントはケーパビリティによって識別され保護される連続バイト列である。セグメントのバイトをプロセスのスレッドにアクセス可能にするために、セグメントはプロセスのアドレス空間のリージョンにマッピングされる。セグメントは同時に複数のリージョンにマッピングすることができ、それらのリージョンは異なるアドレス空間にあってもよい。一度マッピングされると、スレッドはリージョン内のアドレスを読み書きするだけで、セグメントのバイトにアクセスする。セグメントをリージョンにマッピングするために、セグメントがリージョンと正確に同じサイズである必要はないことに注意してほしい。セグメントがリージョンよりも大きい場合、リージョンと同じサイズのセグメントの部分だけがアクセスできる。セグメントのアクセス不可能な部分については、その部分をリージョンに再マッピングすることでアクセス可能にすることができる。一方、セグメントがリージョンよりも小さい場合、マッピングされていないアドレスを読んだ結果はマッパに依存する(マッパは Mach の外部メモリ管理者に似ており、後で説明する)。この場合、マッパは例外をあげたり、0 を返したり、セグメントを拡張したりするように設計することができる。セグメントをリージョンにマッピングすることなく、セグメントのバイトにアクセスするもう 1 つの方法は、ファイルに対する I/O

操作として使われている、従来のシステム呼び出しを使うことである。

5. **ポート** (port):ポートは方向のない通信チャンネルであり、論理的に上限のあるメッセージキューである。各ポートは1つのプロセスに属しており、1つのプロセスだけがそのメッセージを読むことができる。別のプロセスと通信するために、プロセスはメッセージを受信プロセスのポートへ送信する。メッセージは受信側がポートから受け取るまで、そのポートにキューイングされる。Machの場合と同様に、ポートをプロセス間で移動させることができる。また、ポートをグループ化してポートグループを作成することができる。

6. **メッセージ** (message):Chorusでは、基本的な通信パラダイムはメッセージパッシングである。したがって、2つのプロセスは互いにメッセージを交換することで通信を行う。メッセージは受信プロセスのポートへ届けられる。メッセージには、ヘッダ、オプションの固定長部分(64バイト)、オプションの可変長サイズのボディ(最大サイズは64Kバイト)が含まれる。カーネルにとって、固定長部分とボディは型のないバイト配列である。したがって、メッセージの内容に関するセマンティクスはユーザアプリケーションが決めることができる。

7. **ユーザ識別子 (UI)**(user identifier):Chorusでは、ポートやプロセスなどの多くのカーネル資源は64ビットのUIが割り当てられている。この値はシステムにおいて、システムの全生存期間中はグローバルに一意であることが保証されている。この一意性のため、UIは3つの欄から形成されている。すなわち、(a) UIが作成されたマシンのID、(b) システムがリブートするたびにインクリメントされるエポック番号、(c) そのエポックにおいて有効なカウンタである。UIの最初の欄は、対応するオブジェクト(資源)の位置を決めるオブジェクト位置特定機構によってヒントとして使われる。UIは自由にメッセージやファイルによってプロセスからプロセスへと渡すことができる。

8. **ローカル識別子 (LI)**(local identifier):システムワイドに一意であるUIは使用するには長く高価なものである。したがって、効率のため、1つのプロセス内の資源(スレッドやポートなど)は32ビット整数のLIによって識別される。LIはそれを使うプロセス内でのみ有効である。

9. **ケーパビリティ**(capability):資源識別子と資源へのアクセスを制限する両

方のためにケーパビリティが使われている。ケーパビリティは主にサブシステムによって管理されている資源を識別するために使われる。しかし、カーネルによって管理されている資源を識別するために使われる場合もある。ケーパビリティは次の 2 つの欄から成る。(a) 通常はポートの識別子である 64 ビットの UI。このポートにはオブジェクトに対する操作を要求するためにメッセージが送られる。(b) 64 ビットのキー。キーには 2 つの欄がある。最初の欄は資源テーブルのインデックスとして機能する。これによって、同じポートを通じてアクセスされる複数の資源の中から資源を識別することができる。二番目の欄には乱数があり、有効なケーパビリティを推測するのを難しくしている。プロセスはケーパビリティを別のプロセスへ送ることができたり、ファイルの中に格納することができる。

10. **保護識別子 (PI)**(protection identifier):認証目的のために、Chorus における各プロセスは関連する PI を持っている。PI は関連するようなセマンティクスを持たない小さな文字列である。デフォルトで、プロセスの PI はそれを作成したプロセスの PI である。しかし、カーネルやシステムプロセスによって変更することができる。プロセスがメッセージを受信する場合、そのメッセージを送ったプロセスの PI を特定するために、カーネルに要求を行うことができる。この機構はサーバプロセスの設計で使うことができ、これによって、サーバプロセスが管理する資源に対するアクセス制御を実装することができる。例えば、UNIX の保護セマンティクスをエミュレートするために、UNIX ユーザ識別子 (UID) と同等のものを各プロセスに関連づけることができる。そして、Chorus の PI を使ってこの UID を実装することができる。

12.5.4　プロセス管理

12.5.3 章で説明した鍵となる抽象概念の中で、プロセス管理に対して使われている 2 つの基本的な抽象概念は、プロセス (アクタ) とスレッドである。

プロセスのタイプ

Chorus では、プロセスは次のタイプに分類できる。

1. **カーネルプロセス**: すべてカーネルプロセスはカーネル空間にあり、互いに、そして、マイクロカーネルと同じアドレス空間を共有している。カーネル

プロセスはカーネルモードで実行する。カーネルプロセスは信頼されるものであり、マイクロカーネルに対して直接的に呼び出しを行うことが許されている。また、カーネルプロセスには特権が与えられており、I/O を実行したり、システム資源に直接アクセス行う保護された命令を実行することが許されている。高性能のために、Chorus は軽量 RPC の機能を提供している。カーネルプロセスはこれを使ってお互いに通信を行うことができる。この通信機能は他の 2 つのタイプのプロセスでは利用できない。

2. **システムプロセス:** システムプロセスはユーザ空間にあり、ユーザモードで実行する。各システムプロセスは自分自身のアドレス空間を持っている。システムプロセスは特権は与えられていないが信頼はされている。すなわち、システムプロセスはマイクロカーネルを直接的に呼び出しすることができるが、システム資源に直接アクセスする特権命令を実行することはできない。

3. **ユーザプロセス:** ユーザプロセスはユーザ空間にあり、ユーザモードで実行する。各ユーザプロセスにはそれ自身のアドレス空間がある。ユーザプロセスには特権も与えられていなければ、信頼もされていない。したがって、最も力のないプロセスである。

Chorus における 3 つのタイプのプロセスの特徴を図 12.13 にまとめる。

プロセスの種類	空間	実行モード	私的アドレス空間	特権	信任
カーネルプロセス	カーネル	カーネル	無	有	有
システムプロセス	ユーザ	ユーザ	有	無	有
ユーザプロセス	ユーザ	ユーザ	有	無	無

図 **12.13** Chorus における 3 種類のプロセス

プロセスとスレッドの状態

Chorus において、プロセスは次の状態の 1 つになることができる。

1. **アクティブ** (active): プロセスのスレッドが状態を変えることができ、実行するためにスケジュールすることができるとき、プロセスはアクティブな状態にある。

2. **停止** (stopped): プロセスが凍りついたとき、プロセスは停止状態にある。停止状態のプロセスに属するすべてのスレッドは停止状態である。

一方、スレッドは次の状態の 1 つである。

1. **アクティブ** (active): アクティブな状態のスレッドは、あるプロセッサで実行中であるか、実行の候補であり、プロセッサの割り当てを実行キューで待っている。

2. **待機** (waiting): スレッドがブロックし、あるイベントが起きるのを待たなければならない場合、そのスレッドはイベントが起きるまで待機状態に入れられる。

3. **一時停止** (suspended): 一時停止状態にあるスレッドは、プロセッサで実行中でもなければ、イベントが起きるのを待っているわけでもない。別のスレッドか、それ自身のどちらかがスレッドを一時停止するカーネル呼び出しを発行し、意図的に一時停止状態になっている。

4. **停止** (stopped): プロセスに属するすべてのスレッドは、プロセスの状態がアクティブから停止に変わるとき、停止状態に入る。1 つのスレッドが同時に、最初の 3 つの状態のうちの 1 つと四番目の状態になってもよいことに注意してほしい。

次のシステム呼び出しはプロセスとスレッドの状態に影響する。

■ actorCreate は新しいプロセスを作成するのに使用する。プロセスタイプ (カーネル、システム、ユーザ) とプロセスの初期状態をパラメータとして指定することができる。プロセスのケーパビリティが呼び出し側へ返される。

■ actorStop はプロセスの状態をアクティブから停止状態へ変えるのに使用する。このプロセスのすべてのスレッドの状態が停止状態に変わる。

■ actorStart はプロセスの状態を停止からアクティブに変えるために使用する。このプロセスのすべてのスレッドは停止状態からもとの状態を再開する。

■ threadCreate はスレッドを作成するのに使用する。初期状態、優先度などはパラメータとして指定できる。

■ threadSuspend はスレッドを一時停止するのに使用する。

■ threadResume は一時停止状態にあるスレッドを再開するのに使用する。

スレッドの同期

プロセスのすべてのスレッドは共通のアドレス空間を共有するので、スレッドが臨界領域で実行するには、相互に排他制御を行わなければならない。この目的のために Chorus で使われている 2 つの機構は、ミューテック変数の技法と、計数型セマフォの技法である。ミューテック変数の技法 (第 8 章で説明した) は相互排他に対してのみ使う。これには、呼び出し側をブロックさせない操作を、カーネル呼び出しをする必要なく、呼び出し側の空間で実行できるという利点がある。一方、計数型セマフォの技法は、スレッドの一般的な同期に対して使われている。この技法では、UP 操作及び DOWN 操作を実行するためのシステム呼び出しが提供されており、それぞれ計数型セマフォの値をインクリメント、ディクリメントする。Chorus でスレッドの同期に対して提供されているシステム呼び出しは次の通りである。

■ mutexInit はミューテック変数を初期化するのに使用する。

■ mutexGet はミューテック変数に対するロック操作を実行するのに使用する。

■ mutexRel はミューテック変数に対するアンロック操作を実行するのに使用する。

■ semInit は計数型セマフォを初期化するのに使用する。

■ semP は計数型セマフォに対する DOWN 操作を実行するのに使用する。

■ semV は計数型セマフォに対する UP 操作を実行するのに使用する。

スレッドスケジューリング

Chorus において、スレッドは動的に変更することのできる個々のカーネル優先度に従って、カーネルによってスケジューリングされる。各プロセスには

優先度が割り当てられ、各スレッドにもまたそのプロセス内での相対的な優先度が割り当てられる。スレッドの絶対的な優先度はそれ自身の優先度とプロセスの優先度の合計である。アクティブなスレッドのキューが絶対優先度ごとに別々に管理されている。CPUが空いた場合、空でない最も高い優先度の最初のスレッドがその実行に割り当てられる。

Chorusはリアルタイムアプリケーションのサポートを提供していることを説明した。このため、閾値である優先度レベルを使ってアクティブスレッドを2つのグループに分け、異なるスケジューリングアルゴリズムを使って2つのグループのスレッドをスケジューリングしている。すなわち、閾値よりも低い優先度レベルを持つキューに属するスレッドは時分割化され、クオンタムに基づいてラウンドロビンでCPU時間を消費する。一方、閾値よりも高い優先度をもつキューに属するスレッドは、一度実行すると、自発的にCPUを解放するか、さらに高い優先度のスレッドが実行するためにアクティブになるまで、実行し続ける。この機構は多くのリアルタイムアプリケーションを処理するのに十分であると考えられる。

動的に優先度を変えるシステム呼び出しを次に示す。

■ actorPriorityはプロセスの優先度値を読むか、あるいは、変更するために使用する。

■ threadPriorityはスレッドの相対的な優先度値を読むか、あるいは、変更するために使用する。

例外処理

Chorusには、1つの例外ハンドラのカーネルスレッドが存在し、各プロセスもまたそれ自身の例外ハンドラのスレッドを持つことができる。例外が起きた場合、最初にカーネルの例外ハンドラスレッドを使ってそれを処理しようとする。これが失敗した場合、カーネルはその例外を発生したスレッドを一時停止して、それと同じプロセスにある例外ハンドラスレッドにメッセージを送信する。この方法もまた失敗した場合、例外を発生させたスレッドはカーネルによって殺される。

12.5.5　メモリ管理

リージョンとセグメント管理

12.5.3で説明した抽象概念のうち、メモリ管理に使われている2つの基本的な抽象概念は、リージョンとセグメントである。それらについては12.5.3章で既に説明した。したがって、それらの管理と利用に対してChorusでサポートされているシステム呼び出しについてのみ、ここでは説明を行う。以下に重要なシステムのみをあげる(すべてではない)。

■ rgnAllocateはプロセスのアドレス空間にリージョンを割り当て、その属性を設定するのに使用する。プロセスのケーパビリティ、リージョンの開始アドレス、リージョンのサイズ、様々な初期化オプションがパラメータとして指定される。

■ rgnInitはリージョンを割り当て、パラメータで指定されたケーパビリティを持つセグメントでその中身を初期化する。rgnInitに似ているが、割り当てられたリージョンを異なる方法で埋める他の幾つかの呼び出しが提供されている。

■ rgnSetProtectはリージョンに関連した保護の状態(読み取り、書き込み、実行ビット)を変更するために使用する。また、カーネルに対してのみリージョンをアクセス可能にするためにも使われる。

■ rgnStatはリージョンに関するサイズとその他の情報を獲得するために使用する。

■ rgnFreeはリージョンに割り当てられた空間を解放するために使用する。解放された空間は新しいリージョンに割り当てて使うことができる。

■ sgReadはセグメントからデータを読むのに使用する。読もうとするデータの部分はオフセットとバイト数によってパラメータとして指定する。データをコピーする先のバッファアドレスもまたパラメータとして指定する。

■ sgWriteはデータをセグメントに書き込むのに使用する。書く部分はオフセットとバイト数でパラメータとして指定する。データをコピーするバッファアドレスもまたパラメータとして指定する。

マッパ

Chorusのマッパは Machの外部メモリ管理者に非常に似ている。リージョンにマッピングされる各セグメントに対して、それを制御するマッパが存在する。1つのマッパは複数のセグメントを制御することができる。カーネルはマッピングされたセグメントのキャッシュ管理者として働く。すなわち、各カーネルの仮想メモリ要素は、メモリ中のページのページキャッシュを管理し、どのページがどのセグメントに属しているかを追跡している。仮想メモリの構成要

素は、どのキャッシュされたページがダーティで、どれがクリーンかもまた追跡している。キャッシュに新しいページのための空きを作るために、あるページが置き換え対象として選択されると、クリーンならばカーネルは単にそれを破棄する。そうでない場合、それは適切なマッパに送られ、対応するセグメントに書き戻される。一方、ページフォールトが起きた場合、カーネルは最初にページがキャッシュにあるかどうか検査する。もしない場合、カーネルは適切なマッパにそのページを送るよう要求し、フルトしているスレッドをページが受け取るまで一時停止する。そして、マッパはページがまだアドレス空間になければ、セグメントの記憶装置場所からページを取り出す。マッパはページが利用可能になるとカーネルに通知を行う。そして、カーネルはマッパからのページを受け取り、メモリ管理ユニット (MMU) のページテーブルを更新して、フォールトしているスレッドを再開する。

Chorus において提供される幾つかのマッパ関連のシステム呼び出しを次に示す。

■ MpCreate はカーネルやプログラムがマッパに対して、セグメントをスワップアウトするように要求したり、ディスク領域を割り当てるように要求するために使用する。この呼び出しに対する応答として、マッパはディスクに新しいセグメントを割り当て、呼び出し側にそれに対するケーパビリティを返す。

■ MpRelease はマッパに対して既に作成されているセグメントを解放するように要求するのに使用する。

■ MpPullIn はカーネルがマッパに対してそれが制御しているセグメントからデータを要求するのに使用する。

■ MpPushOut はカーネルがマッパに対してマッパが管理しているセグメントのデータを送るのに使用する。カーネルがキャッシュからセグメントダーティページを置き換えようとするときにこれが必要となる。

分散共有メモリ

Chorus でサポートされている分散共有メモリの機構は次に示す事柄によって特徴づけることができる (これらの特徴に関する詳細は 5 章を参照せよ)。

1. ブロックサイズと同じページサイズを使用する。これを機能させるため、セグメントは 1 つ又は複数のページからなるフラグメントに分割される。

2. 複製された、移動するブロック (RMB) による戦略をとっている。

3. メモリの一貫性の問題を単純化するために、フラグメントを2つのタイプに分類している。読み取り専用と読み取り書き込みである。読み取り書き込みフラグメントの複製は行えない。したがって、読み取り専用のフラグメントのみが複製され、読み取り書き込みのフラグメントはシステム全体で1つのコピーしかない。

4. データ位置特定に対して動的な分散サーバのアルゴリズムを使用している。

12.5.6 プロセス間通信

12.5.3で説明した鍵となる抽象概念のうち、プロセス間通信に対して使われている2つの基本的な抽象概念はメッセージとポートである。メッセージはポートに対して送受信される。ポートはプロセスによって動的に作成そして削除される。ポートの作成と削除に対するシステム呼び出しを次に示す。

■ portCreateはポートを作成するために使用する。ポートのケーパビリティが呼び出し側に返される。このケーパビリティを別のプロセスに送り、それによって、そのプロセスがこのポートに対してメッセージを送ることが可能となる。

■ portDeleteはポートを削除するのに使用する。

メッセージの送受信

Chorusはスレッドが柔軟な方法でメッセージを送受信できるように次の通信操作を提供している。

1. **非同期送信**:ipcSendシステム呼び出しはポートに対して非同期にメッセージを送信するのに使用する。メッセージ配送が成功するという保証はなく、通信障害が起きた場合、送信側に通知は行われない。このタイプの通信機能は、ユーザがChorusの基本的な通信機能の上に任意の通信形態を構築するのに使うことができる。

2. **受信**:ipcReceiveシステム呼び出しはメッセージを受信するために使用する。この場合、2つのオプションが利用できる。最初のオプションでは、スレッドは受信したいポートを指定できる。二番目のオプションでは、プロセスは所有しているポートの任意の1つから受信したいということを指定できる。二番

目のオプションを持つ受信操作は、次の機構を使って、さらに選択可能なものにできる。

(a) portDisable システム呼び出しを使ってポートの幾つかを使用不可とすることで、利用可能なポートだけが要求を満たす候補となる。使用不可となったポートは、portEnable システム呼び出しを使って使用可にできる。ポートは動的に使用可そして不可にできる。

(b) ポートに優先度を割り当てることで、1 つより多くの使用ポートにメッセージがある場合、最も高い優先度を持つ使用ポートを選択してメッセージを受信できる。ポートの優先度は動的に変えることができる。

ipcReceive システム呼び出しを実行したときにメッセージが利用できない倍、呼び出しスレッドはメッセージが到着するか、ユーザが指定した時間が切れるまで、一時停止される。

Chorus は効率のために Mach の書き込み時コピー機構を使っている。したがって、メッセージが受信された場合、送信側と受信側が異なるマシンにあるときのみ、メッセージの本体が受信プロセスのアドレス空間にコピーされる。2 つが同じマシンにある場合、メッセージの本体は単に受信プロセスのアドレス空間のリージョンにマッピングされるだけである。そして、この場合、受信側がメッセージの本体を修正したとき、メッセージの本体がコピーされる。

3. **RPC**:Chorus でサポートされている三番目のタイプの通信は RPC である。RPC によって要求/応答タイプのインタラクションが可能となる。RPC を起動するシステム呼び出しは ipcCall である。この呼び出しを実行するプロセスは応答が返ってくるか、RPC のタイマが切れるまで、自動的にブロックする。そして、どちらかが行われると送信側のブロックがとける。RPC に対して最大一回セマンティクスがサポートされている。

グループ通信

グループ通信の機能が Chorus ではサポートされており、これによって複数のポートをグループ化して**ポートグループ** (port group) を形成することができる。このために提供されているシステム呼び出しを次に示す。

■ grpAlloc は空のポートグループを作成するのに使用する。新しく作成されたポートグループのケーパビリティが呼び出し側へ返される。このケーパビリティを使って、呼び出し側か、引き続いてこのケーパビリティを獲得したプロセスは、ポートグループに対してポートを作成又は削除することができる。

■ grpPortInsert は既存のポートグループに新しいポートを加えるために使用する。

■ grpPortRemove はポートグループからポートを削除するために使用する。

ポートグループに対してメッセージを送信する送信側は柔軟に次のアドレスモードの1つを選択することができる。

1. 放送モード (broadcast mode):この場合、メッセージはグループのすべてのポートに送信される。この通信モードは、ファイルの更新をその複製を持っているすべてのサーバに送ったり、サーバグループのどのメンバかはわからないが、幾つかのメンバが管理している資源に対して操作を要求するような場合に便利である。前者の例では、グループのすべてのメンバがメッセージを受信したときに行動をとり、後者の例では、資源を管理しているメンバだけがメッセージを受信したときに行動をとり、その他のメンバは単にメッセージを捨てる。

通信の放送モードはメッセージ順序の保証を提供していないことに注意してほしい。これが必要な場合、それはユーザが実装する必要がある。

2. 機能モード (functional mode):この場合、メッセージはシステムによって選択されたグループ内の1つのポートにだけ送られる。この通信モードは、たくさんのサーバによって同様にサービスが実行され、クライアントはどのサーバがサービスを提供したかについて気にしない、あるいは、クライアントが特定の割り当てられたサーバの区別に関する知識に悩まされようとは思わない場合に便利である。そのような場合、クライアントはサービスを提供するサーバグループに単に要求を送り、システムはクライアントの要求にサービスするための最も適したサーバを選択する。

ファンクショナルモードの通信を行う趣味深いアプリケーションは、システムの再構成可能なサービスを提供すことである。例えば、あるサービスを提供するすべてのサーバを1つのポートグループにグループ化することができる。そこで、クライアントは要求を機能モードでそのグループへ送る。この時、クライアントはそのサービスを提供するのにどのサーバがその時点で利用できるかに関して知る必要はない。この透過性の特徴によって、更新されたサーバをグループに追加したり、古いサーバをグループから削除できる。このとき、サービスを中断したり、クライアントがシステムが再構成されたということを

意識する必要はない。

3. 選択的機能モード (selective functional mode):この場合もまた、メッセージはグループの 1 つのポートだけに送られる。しかし、ポートはシステムではなく呼び出し側によって選択される。この通信モードの実際的な利用は負荷分散にある。過負荷なノードは最初に放送通信モードを使って他のすべてのノードの現在の負荷を獲得する。受け取った応答から、最も軽い負荷のノードを選択する。そして、選択的機能通信モードを使って、負荷の幾らかを選択したノードへ転送する。

グループ通信の 3 つのモードを図 12.14 にまとめる。3 つのすべてのモードにおいて、メッセージは非同期送信によって送られ、したがって、通信は信頼性のないものであることに注意してほしい。

通信モード	メッセージ送信先	メンバの選定者
ブロードキャスト	すべてのメンバ	―
ファンクショナル	一人のメンバ	システム
選択的ファンクショナル	一人のメンバ	呼び出し側

図 **12.14**　Chorus のグループ通信モード

ポート移動

Chorus では、portMigrate という名前のシステム呼び出しを使ってポートをあるプロセスから削除して、別のプロセスへそれを移動することができる。ポートが移動させられるとき、そこにあるすべてのメッセージも一緒に移動する。期待するように、ポートは移動したときもポートグループのメンバのままである。

ポート移動機能によって、あるサービスを提供する責任を動的にあるサーバから別のサーバへと転送することができる。この際、そのサービスに対するサーバと現在インタラクションを行っているクライアントは気づかない。この機能を持っ趣味深いアプリケーションは、保守透過性をサポートするためのものである。例えば、メインテナンスのために停止するマシンのサーバは、そのポートを同じサービスを提供する異なるマシン上の別のサーバに移動すること

ができる。クライアントの要求は何ら中断することなく、一貫して処理しつづけられる。このように、サーバマシンをメインテナンスのために停止でき、クライアントが気づくことなく後で起動することができる。

ネットワークワイドな IPC

ネットワークワイドな IPC のために、Chorus は Mach のネットワークメッセージサーバに似た**ネットワーク管理者** (network manager) を使っている。ネットワーク管理者はシステムの各ノードにあり、すべてのネットワーク管理者が一緒に動作して、カーネルの通信機能をネットワークに渡って透過なものに拡張する。

スレッドがポートにメッセージを送る場合、そのマシンのカーネルがローカルポートのリストからそのポートの UI を検索する。もしリストになかった場合、カーネルはポートの UI をローカルなネットワークワーク管理者に転送する。ローカルなネットワーク管理者は他のネットワーク管理者と通信を行い、そのポートの場所を獲得する。一度、ポートの場所がわかると、その後そのポートに送られるメッセージは、メッセージポートのあるノードのネットワーク管理者のポートへ直接送られる。そのノードのネットワーク管理者はメッセージを送られる先のローカルポートへ転送する。

12.5.7 UNIX エミュレーションと拡張

Chorus は Chorus/Mix(Modular UNIX) と呼ばれるサブシステムを構築した。これは、UNIX System V とバイナリ互換性を提供する。UNIX エミュレーションサブシステムの設計で使用されている基本的なアプローチを簡単に次に説明する。更に詳しいことは [Gien and Grob 1992, Armand et al. 1989] に見ることができる。

Chorus/MiX サブシステムは主に次のプロセスから構成されている (図 12.15 を見よ)。

1. **プロセス管理者** (process manager):このプロセスはエミュレーションサブシステムの中心的な構成要素である。これはすべてのシステム呼び出し割り込みを捉え、必要があれば、システム呼び出しを処理するためにサブシステムの他のプロセスと通信を行う。更に、それはプロセス管理に関係する作業 (プロセスの作成と終了)、シグナル処理、資源名前付けを実行する。

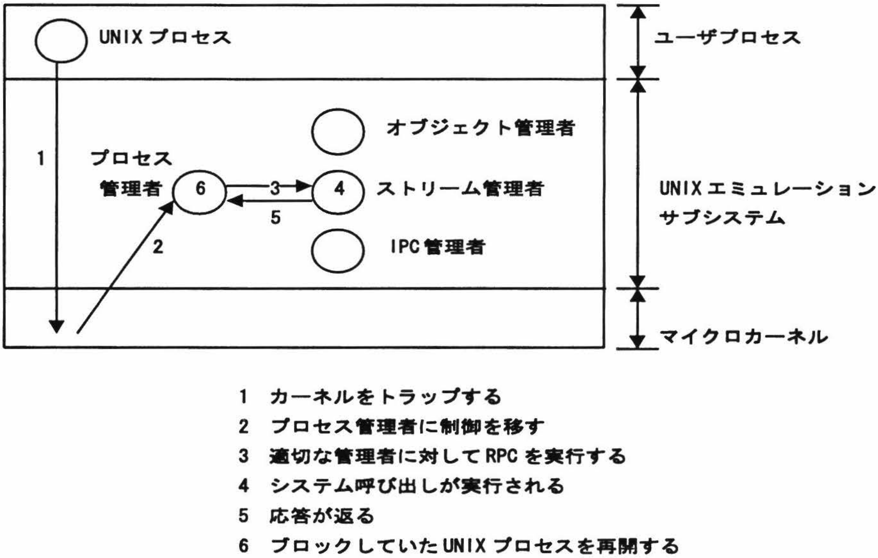


図 **12.15**　Chorus の UNIX エミュレーション

2. オブジェクト管理者 (object manager):このプロセスは主にファイル管理の作業を行い、ディスクドライバを持つこともできる。これは制御を行っているファイルのマッパとしても働く。

3. ストリーム管理者 (stream manager):このプロセスは、パイプ、ソケットとネットワーキング、そしてキーボード、ディスプレイ、マウス、テープデバイスなどのストリームデバイスを管理する。

4. プロセス間通信管理者 (interprocess communication manager):このプロセスは System V メッセージ、セマフォ、共有メモリに関連したシステム呼び出しの処理を行う。

これら 4 つのプロセスのうち、プロセス管理者には、大量の新たに書かれたコードがある。一方、他はほとんど UNIX コードそのものしかない。さらに、プロセス管理者のみが、UNIX エミュレーションサブシステムのあるすべて

ノードに必要である。他のプロセスはオプションであり、必要なときのみロードされる。例えば、オブジェクト管理者はディスクレスのノードではロードされない。

4つのプロセスはカーネルモードかあるいはユーザモードのどちからで実行することができる。しかし、性能とセキュリティを理由として、現在はすべてがカーネルモードで実行されている。

プロセス管理者には複数のスレッドがある。他の管理者は初期は1つのスレッドで開始し、要求が来るにつれて追加のスレッドが作成される。

システムがブートされる場合、システムは環境を調べ、それに従って必要とされるオプションのプロセスをロードする。その初期化の一部として、これらのオプションのプロセスのそれぞれが、プロセス管理者にメッセージを送信し、そのポートをアナウンスし、どのようなシステム呼び出しを処理できるかについて通知する。一方、プロセス管理者は、UNIXがシステム呼び出しを実行するのに使うトラップ番号を自分が処理したいことをカーネルに伝える。カーネルはエミュレートされる各システム呼び出しに対してルーチンのアドレスを持つテーブルを管理する。

この設定によって、UNIXプロセスがシステム呼び出しを実行し、カーネルをトラップした場合、対応するシステム呼び出しを処理するプロセス中のスレッドが自動的にカーネルから制御を獲得する。システム呼び出しが完全にローカルに処理できる場合、プロセス管理者は要求されたシステム呼び出しそのものを実行する。そうでない場合、システム呼び出しにも依存するが、プロセス管理者はRPCをオブジェクト管理者か、ストリーム管理者か、プロセス間通信管理者かに行う。コンタクトされた管理者プロセスは、要求されてシステム呼び出しを実行し、応答をプロセス管理者に返す。そして、プロセス管理者は適切な返り値を設定してブロックしていたUNIXプロセスを再開する。この全過程をまとめたものが図12.15である。図に示されている例では、システム呼び出しはストリーム管理者によって処理されている。

MachのUNIXエミュレーション方法との比較

UNIXエミュレーションのサポートに対して、MachとChorusでとられている方法の比較による利点と欠点について次に示す [Coulouris et al. 1994]。

1. モジュール性:Chorusの設計はMachの設計よりもモジュール性が高い。なぜなら、エミュレーションサブシステムは異なるUNIXの機能を提供するた

めに異なるサーバを使用している。これとは反対に Mach では、1 つの UNIX サーバがすべての UNIX 機能を提供するために使われている。Chorus の 4 つの管理者プロセスはどの変数も他のメモリを共有せず、しかも、専ら RPC によって通信するため、どの管理者プロセスも独立に再実装することができ、提供されるインタフェースは変えないで済む。

2. **システム状態の管理**:Chorus が複数のサーバプロセスを使っているのに対して Mach は 1 つの UNIX サーバプロセスを使っているため、Mach ではシステム状態を管理するのがより簡単である。これは、Chorus では、1 つのエミュレートされて UNIX プロセスに関係するシステム状態が、異なるマシンに存在することのできる幾つかのサーバプロセスに分散しているからである。この状態が性能のために複製される場合、機構と余分な通信オーバヘッドが状態の一貫性を保つために必要になるであろう。一方、状態は分散しているが、複製されていない場合、機能は影響を受けやすい。なぜなら、互いに依存することなく異なるサーバが障害になり得るからである。このような問題は Mach の方法には関係しない。なぜなら、プロセスの状態はエミュレーションライブラリのデータと 1 つの UNIX サーバプロセスに閉じ込められるからである。

3. **保護**:Mach では、プロセスが UNIX システム呼び出しを実行し、カーネルにトラップをあげる場合、制御はプロセスのアドレス空間にあるエミュレーションライブラリに戻される。したがって、プロセスにエミュレーションライブラリのデータを勝手に触ってしまうバグのあるコードがあった場合、標準でない障害を引き起こし問題を発する可能性がある。Chorus で使われているアプローチは、完全にシステムデータ構造とユーザプロセスを隔離するので、そのような問題を避け、Mach のアプローチよりもシステムに対するより優れた保護を提供する。

UNIX エミュレーションに対して Mach と Chorus によってとられている方法の違いに関する詳細は [Dean and Armand 1992] に見ることができる。

UNIX に対する拡張

UNIX エミュレーションに加えて、Chorus は UNIX に対する拡張も提供しており、これによって UNIX プロセスが UNIX では利用できない標準的な Chorus の特性を使うことができる。最も重要なものの幾つかを次に示す。

1. スレッド:この拡張によって、UNIX プロセスは Chorus スレッドパッケージを使って新しいスレッドを作成したり、破棄したりすることができる。

2. 遠隔プロセス作成:この拡張によって、UNIX プロセスが新しいプロセスを遠隔のノードに作成することができる。プロセス作成のためのシステム呼び出しでは、新しいプロセスをローカルなノードではなく指定した遠隔のノードで作成するように指定できる。新しいプロセスはフォークされたときはローカルなノードで開始するが、exec システム呼び出しを実行したときに、指定した遠隔のノードで開始される。

3. グループ通信:Chorus と同じグループ通信機能が UNIX プロセスにも享受できる。なぜなら、UNIX サブシステムを使っているユーザプロセスは Chorus のプロセスが行うのと同様にポートとポートグループを作成し、メッセージを送受信できるからである。

4. メモリモデル:UNIX プロセスは Chorus でサポートされているメモリモデルの柔軟性を享受できる。すなわち、UNIX サブシステムを使っているユーザプロセスは、Chorus のプロセスと同様にリージョンを作成し、それに対してセグメントをマッピングできる。

12.5.8 COOL サブシステム

Chorus/MiX サブシステムに加えて、Chorus のために開発されたもう 1 つのサブシステムは Chorus Object-Oriented Layer(COOL) である。その主な目標は、細粒度のオブジェクト指向言語とアプリケーションに対するシステムレベルのサポートを Chorus で提供することである。このサブシステムを設計するのに使われた基本的な方法を簡単に以下に説明する。更に詳しいことは [Lea et al. 1991, 1993] に見ることができる。COOL(COOL-2、第 2 版) サブシステムは Chorus マイクロカーネルの上に位置し、次の層に構造化されている (図 12.16 参照)。

1. COOL 基本層:この層はシステム呼び出しインタフェースを提供し、これによってユーザアプリケーションに対して新しいオブジェクト指向のマイクロ

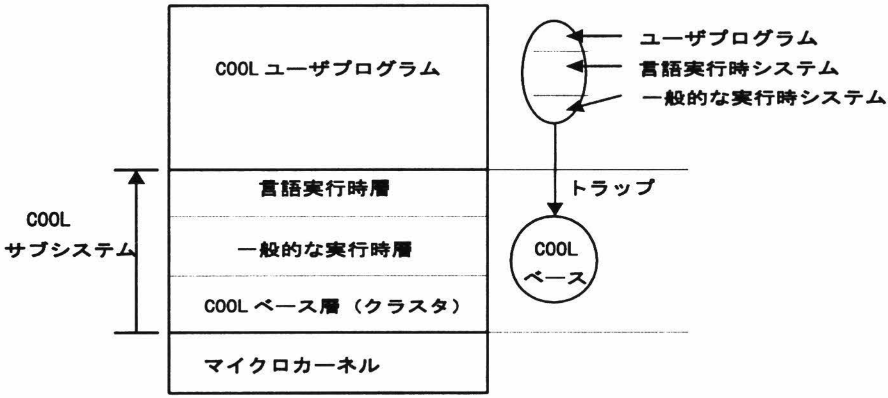


図 12.16　Chorus の COOL サブシステムの実装の構造と層

カーネルの幻想を与える。この層は**クラスタ** (cluster) と呼ばれる抽象概念を扱い、クラスタはオブジェクトが存在する場所である。クラスタは同じクラスに属しているオブジェクトのような、関連する一群のオブジェクトを所有している。どのオブジェクトがどのクラスタに属するかに関する決定は、より上位の層によって行われる。

メモリモデルの観点から、クラスタはセグメントを背後にもつ Chorus リージョンの集合である。したがって、クラスタは異なるマシンに存在し得る複数のプロセスのアドレス空間に同時にマッピングできる。しかし、クラスタは複製されず、どの時点でも各クラスタには 1 つの物理的なコピーしか存在しない。遠隔のクラスタに対応する要求は、そのクラスタを現在所有しているマシンに対して遠隔起動の要求を転送するか、クラスタを要求を行っているマシンに移動してローカルな起動を行うかによってサービスされる。このように、COOL 基本層はクラスタを管理し、クラスタを複数のアドレス空間にマッピングしてすべての COOL プロセスに対してそれがどこで実行されていようが関係なく可視な分散クラスタ空間を生成する。

2. **一般的な実行時 (GRT) 層**:この層はクラスタ内の細粒度のオブジェクトに対するサポートを提供する。特に、オブジェクトを作成したり、削除するための操作や、オブジェクトをアドレス空間にマッピングしたり外したりするための操作や、オブジェクトのメソッドを起動するための操作を提供する。また、

単一レベルの永続オブジェクトの格納、Chorus RPC に基づくオブジェクト間通信、アプリケーションが実行している間のオブジェクトの保護も提供する。

3. **言語実行時層**:プログラマはオブジェクトを定義するために、C++、Smalltalk、Eiffel などの異なるオブジェクト指向プログラミング言語を使うことができる。言語に固有の実行時層は、そのような幾つかのプログラミング言語のオブジェクトモデルを GRT の抽象概念にマッピングする。その実行時層はプリプロセッサを使って、GRT レベルで作成されるオブジェクトのそれぞれのタイプに対して上位呼び出しを生成する。この機構は、一般的な実行時層と言語依存の実行時層の標準インタフェースを構築するのに使用される。一般的な実行時層はこのインタフェースを使って言語実行時層に対する呼び出しを実行し、ある操作のセマンティクスに関する言語固有の情報を獲得する。例えば、インメモリのオブジェクトポインタを格納するために永続ポインタにどのように変換すれば良いか、また、メソッドディスパッチをどのように処理すれば良いかについて知る。この機構によって、COOL は理に適った効率性によって、多くの異なるオブジェクト指向プログラミング言語のサポートが可能になっている。

これら 3 つの層のうち、COOL 基本層はサブシステム層で実行する Chorus プロセスとして実装されている。一方、一般的な実行時層のソフトウェアはそれぞれの COOL プログラムにリンクされ、言語固有の実行時層の適切なソフトウェア (COOL プログラムが書かれている言語に依存する) が COOL プログラムにリンクされる。したがって、言語実行時と一般的な実行時システムはユーザプログラムのアドレス空間にある。

12.6 AMOEBA、V-SYSTEM、MACH、CHORUS の比較

これまでに説明した 4 つの分散オペレーティングシステム (Amoeba、V-System、Mach、Chorus) はそれぞれの長所と短所があり、それは設計者たちと進化の歴史によって設定された目標によるものである。Amoeba と V-System は疎結合の分散メモリマルチプロセッサに対する分散オペレーティングシステムとしてゼロから設計された。今日に至るまで、それらのシステムは大学の研究プロジェクトである。一方、Mach は密結合の共有メモリマルチプロセッサ

のためのオペレーティングシステムを構築するために、大学の研究プロジェクトとして始まり、後に疎結合の分散メモリマルチプロセッサシステムのために拡張された。後に、Mach は商用化のために選ばれた。最後に、Chorus もまた疎結合の分散メモリマルチプロセッサのための分散オペレーティングシステムを構築するための、研究プロジェクトとして始まった。しかし、その設計目標はそれが研究プロジェクトから商用のシステムへと発展するに従って変わった。これらのシステムの設計目標と進化の歴史の結果は、システムの長所と短所として表れている。これらの違いをより明らかにするために、これら 4 つのシステムすべてについて、その重要な側面を見ることにしよう。

12.6.1　システムモデル

Amoeba はプロセッサプールモデルに基づいており、一方、V-System、Mach、Chorus はワークステーションモデルに基づいている。したがって、Amoeba では、ユーザは特定のマシンにログオンせず、全体としてシステムにログオンする。そして、ユーザのジョブを動作させる CPU(又は複数の CPU) の選択はオペレーティングシステムによって自動的に行われる。一般に、ユーザはどの CPU は自分のジョブを処理しているかについて意識することはない。

一方、V-System、Mach、Chorus では、ユーザは特定のマシン (このマシンはホームマシンと呼ばれる) にログオンする。ユーザのジョブの多くはデフォルトでそのマシンで実行する。ユーザはそのシステムにある自分のジョブを遠隔実行するための要求を行うことができる。更に、V-System では、自動的な負荷分散の機能も提供している。そのおかげで、ユーザのジョブは知らないところで遠隔で処理することができる。しかし、これらのどの場合においても、3 つのすべてのシステムがジョブのローカルな処理と遠隔の処理という概念を持っている。この概念は Amoeba には存在しない。

12.6.2　カーネル

4 つのすべてのシステムがマイクロカーネルに基づいている。マイクロカーネルでは、カーネルは最小であり、他の機能はユーザレベルのサーバによってサポートされる。しかし、マイクロカーネルの複雑性と 4 つのシステムのカーネルインタフェースは、それらのシステムが提供する柔軟性によって異なる。Amoeba と V-System のマイクロカーネルは単純であり、非常に少数のシステ

ム呼び出ししかない。Mach のマイクロカーネルはかなり複雑であり、非常に多くのシステム呼び出しがある。これは主に、Mach がより優れた柔軟性を提供しようとしていることによる。その柔軟性によって、同じことを 2 つあるいは 3 つの方法で行うことができたり、ユーザは特定の環境に対して最も便利であったり、効率の良い方法を選択するオプションを持つ。Chorus のマイクロカーネルは Mach よりは少ないシステム呼び出ししかないが、Amoeba や V-System よりは多い。したがって、複雑さは中程度である。

12.6.3 プロセス管理

Amoeba、V-System、Mach、Chorus におけるプロセスに対応する抽象概念は、それぞれ、プロセス、チーム、タスク、アクタである。4 つのシステムすべてがマルチスレッド化されたプロセスをサポートしており、すべてのシステムで、スレッドはカーネルによって管理されスケジュールされている。

Mach と Chorus には、異なる CPU で並列に同一のプロセスのスレッドを実行する柔軟性がある。この柔軟性は Amoeba と V-System では利用できない。したがって、これらのシステムでは、1 つの CPU がプロセスのスレッドによって時間共有され、擬似並列的に実行する。

Amoeba はユーザが制御するタイプのスレッドスケジューリングを提供していない。V-System と Chorus は、プロセスがそのスレッドの優先度を設定することが可能である。Mach はこの場合最大の柔軟性を提供している。なぜなら、スレッド優先度を動的に変更できるのに加えて、ハンドオフスケジューリングの機能も提供しているからである。この機能はスレッドが CPU の使用が終わったときにその CPU を選択した別のスレッドに渡すために使うことができる。

Amoeba と V-System は自動的な負荷分散の機能を提供しており、システムのすべてのマシンに統一的な方法で負荷を分散させる。Mach と Chorus はこの機能を持っておらず、デフォルトで、ユーザのジョブはそのホームマシンで実行される。明示的に要求したときのみ、ユーザのジョブを遠隔で実行できる。

12.6.4 プロセス間通信

Amoeba、V-System、Mach、Chorus はすべてプロセス間通信のために RPC 機能を提供している。Mach と Chorus は非同期にメッセージを送信する機能もまた提供している。Mach はメッセージパッシングのすべての形式を 1 つのシ

ステム呼び出しに閉じ込めている。一方、Chorus は選択可能な呼び出しを提供している。Mach がメッセージパッシングのすべての形式を 1 つのシステム呼び出しへ閉じ込めようとした試みによって、通信インタフェースは非常に複雑になっている。なぜなら、非常に多くのパラメータとオプションが可能なすべての場合を処理するために、システム呼び出しの中で使われるからである。

Chorus はローカルなカーネルプロセス間で効率的な通信を行うためにライトウエイト RPC を使用している。他の 3 つのシステムは軽量 RPC は用いていない。

Amoeba と V-System では、メッセージは、固定長のヘッダとオプションである可変長のアウトラインブロックのデータから構成される。Chorus では、メッセージは、固定長のヘッダ、オプションである固定長 64 バイトのメッセージ内データ、そして、オプションである可変長の最大 64 キロバイトのメッセージ内データから構成される。この場合、Mach は最大の柔軟性を提供しており、複数の可変長のメッセージ外のデータブロックを 1 つのメッセージにすることができる。

Mach のメッセージは単純にも、複雑にもなる。単純なメッセージの内容は、バイトの連続列であり、一方、複雑なメッセージは型付きのデータを持つ。複雑なメッセージはケーパビリティを転送するために使うことができる。一方、Amoeba、V-System、Chorus は、単純な型のないメッセージを使用しており、そこにはバイトの連続列が入る。メッセージは Mach と Chorus ではポートに、Amoeba ではプロセスに、V-System では管理者 ID に届けられる。4 つのシステムすべてが、ポートあるいはプロセスを位置づけるためにヒントキャッシュと放送の機構を使っている。

ローカルな IPC では、Mach と Chorus は書き込み時コピーの機構を使い、メッセージ外のデータを渡す。Amoeba と V-System はこの機構を使っていない。したがって、Mach と Chorus のローカル IPC は Amoeba や V-System のものよりも高速である。

Amoeba、V-System、Chorus はグループ通信をサポートしているが、Mach はサポートしていない。Amoeba のグループ通信機能は、信頼性のある、順序づけられた放送機能を提供している。Chorus のそれは、信頼性のない、順序づけられていない放送機能である。そして、V-System は順序づけられていない放送機能を提供しており、ユーザが信頼性の程度を選べる柔軟性を提供している。

Amoeba と V-System では、ネットワークワイドな IPC はカーネルによって処理される。一方、Mach と Chorus では、ユーザレベルのネットワークサーバ

がネットワークワイドな IPC のために使われる。したがって、ノード間の IPC は Amoeba と V-System のほうが、Mach や Chorus よりも高速である。

ネットワークに関して、4 つのすべてのシステムは従来の TCP/IP をサポートしている。さらに、Amoeba は FLIP、そして V-System は VMTP をサポートしている。これらのプロトコルは、標準でもなく、広く使われているものでもないが、分散オペレーティングシステムにおける必要のために特別に設計されており、典型的な RPC に使っている従来のネットワークプロトコルよりも高速である。

12.6.5 メモリ管理

Amoeba は仮想メモリ (デマンドページング) 機構のない非常に単純なメモリ管理方式を持つ。したがって、Amoeba のプロセスが実行するとき、そのアドレス空間全体がメモリの中に存在する。この方法は実装するのに簡単であり、高性能という利点がある。しかし、マシンは極端に大きなメモリを持つことが必要になる。すべてのマシンで最も大きなメモリサイズを持つマシンのメモリを越える大きなアドレス空間のプロセスはそのシステムでは実行できない。対照的に、V-System、Mach、Chorus はページ仮想メモリ機構のサポートを提供している。したがって、アドレス空間がいかに大きいかに関係なく任意のプロセスをシステムで実行できる。

Mach と Chorus の仮想メモリ管理の方式は非常に強力であり柔軟である。なぜなら、それによって、様々方法で複数プロセスの間でページを共有することができるからである。例えば、書き込み時コピーの共有機構によって、1 つのノードにある複数のプロセス間で効率的なページ共有ができる。また、外部ページャ機構によって、仮想メモリを異なるマシンで動作するプロセス間でも共有することが可能になる。

Amoeba、V-System、Mach、Chorus のすべてが分散共有メモリのサポートを提供している。Amoeba はオブジェクト準拠の分散共有メモリを提供しており、そこでは、可変サイズのオブジェクトが、それを使うすべてのシステムに複製されることで共有される。オブジェクトに対する読み取りの操作はローカルに実行される。一方、オブジェクトに対する書き込みの操作は、Amoeba の信頼性のある同報を使って実行される。一方、V-System、Mach、Chorus はページ準拠の分散共有メモリを提供している。Amoeba と同様に、V-System では、読み取り専用のページと書き込み可能なページの両方が複製できる。し

かし、V-System ではメモリの一貫性の問題を解決するために問題指向のアプローチを用いている。一方、メモリの一貫性の問題を解決するために、Mach と Chorus は、読み取り専用のページだけを複製できるようにしており、書き込み可能なページはシステム全体において 1 つのコピーしか存在しない。書き込み可能なオブジェクトの複製とメモリ一貫性のために行う全てに対する書き込みのプロトコルの使用によって、Amoeba 方式による更新は V-System、Mach、Chorus の方式よりも高価なものになっている。しかし、Amoeba の方式とは異なり、Mach、Chorus の方式はスラッシングの問題にを持つ。書き込み可能なページが 2 つのマシンで非常に多くアクセスされると、スラッシングが起こり得る。

12.6.6　ファイル管理

Amoeba のファイル管理方式は V-System、Mach、Chorus のそれとは全く異なる。Amoeba は不変ファイルモデルを用いているのに対して、V-System、Mach、Chorus は可変ファイルモデルを用いている。したがって、Amoeba ではファイルを上書きすることはできず、新しいファイルの版を作成することで更新が行われる。V-System、Mach、Chorus では、ファイルは同じファイルを上書きすることで修正される。

Amoeba では、ファイル全体が常にメモリとディスクの両方に連続した形で格納される。これはファイルに対する効率的なアクセスを提供するが、外部断片化の問題を導いている。V-System、Mach、Chorus では、ファイルは連続して格納される必要はない。記憶領域は固定長のブロックか可変サイズのセグメントを単位として割り当てられる。

4 つのシステムすべてが遠隔ファイルアクセスのためにデータキャッシュモデルを用いている。しかし、Amoeba では、キャッシュの単位はファイルであり、一方、他の 3 つのシステムでは、キャッシュの単位はブロックである。

Amoeba は不変ファイルモデルを使っているので、不変共有ファイルのセマンティクスをサポートしている。一方、V-System、Mach、Chorus のファイル共有セマンティクスは、使用しているユーザレベルのファイルサーバによってサポートされるセマンティクスに依存している。

Amoeba はファイルの自動的な複製もサポートしている。他の 3 つのシステムはこの機能はサポートしていない。

12.6.7 セキュリティ

AmoebaとChorusでは、資源は名前づけられており、ケーパビリティによって保護されている。したがって、これら2つのシステムでは、ケーパビリティ準拠のアクセス制御の機構が使用されている。Machでは、アクセス制御の機構は、ポート権利(ポートケーパビリティ)に基づいている。すなわち、Machのサーバは一般に多くのポートを管理しており、各資源に対して1つのポートがある。そして、そのポートに対して適切なポート権利を所有しているプロセスだけが、対応する資源にアクセスすることができる。最後に、V-Systemでは、アクセス制御の機構は資源管理者によって管理されているACLに基づいている。

AmoebaとChorusのケーパビリティはユーザ空間で管理されており、Machのポートケーパビリティはカーネルで管理されている。Amoebaのケーパビリティは一方向暗号関数を使って保護されている。Chorusもまた保護識別子を提供しており、実際のメッセージの送信側を知るためにこれを使う。

12.6.8 その他の違い

ChorusとV-Systemはリアルタイムアプリケーションをサポートしている。AmoebaとMachはリアルタイムアプリケーションに対して特別な機能を提供していない。

4つのシステムすべてがUNIXエミュレーションをサポートしている。しかし、V-System、Mach、Chorusがバイナリ互換性である一方、AmebaはUNIXとのソースコード互換性をサポートしている。

Chorusは細粒度のオブジェクト指向言語とアプリケーションに対するシステムレベルのサポートもまた提供している。その他の3つのシステムはこの機能は現在提供していない。

図12.17は上記に示した4つのシステムの比較の概要を表している。

12.7 まとめ

この章ではこの本の先の章で説明した様々な概念と実際のシステムを関係づけるために、4つの分散オペレーティングシステムに関する説明を行った。これらのシステムは、Amoeba、V-System、Mach、Chorusである。これらのシ

特徴	Amoeba	V-System	Mach	Chorus
システムモデル	プロセッサプール	ワークステーション－サーバ	ワークステーション－サーバ	ワークステーション－サーバ
カーネルモデル	マイクロカーネル	マイクロカーネル	マイクロカーネル	マイクロカーネル
システム呼び出しインタフェースでのカーネルの複雑性	単純（数個のシステム呼び出し）	単純（数個のシステム呼び出し）	非常に複雑 過多なシステム呼び出し）	かなり複雑（たくさんのシステム呼び出し）
プロセス概念	プロセス	チーム	タスク	アクタ
プロセス内でのマルチスレッドをサポートしているか？	はい	はい	はい	はい
プロセスのスレッドは異なるCPUで動作可能か？	いいえ	いいえ	はい	はい
スレッドが何によって管理、スケジューリングされるか	カーネル	カーネル	カーネル	カーネル
ユーザによるスレッドスケジューリングの制御サポート	いいえ	はい（優先度の割り当て）	はい（優先度の割り当てとハンドオフスケジューリング）	はい（優先度の割り当て）
自動的な負荷分散機能？	はい	はい	いいえ	いいえ
プロセス移動機能？	いいえ	はい	いいえ	いいえ
基本的なIPC機能	RPC	RPC	RPCと非同期送信	RPCと非同期送信
ライトウェイトなRPCを使う	いいえ	いいえ	いいえ	はい。ローカルなカーネルプロセス間での通信のみ。
メッセージ本体	1つの可変サイズの入りきらないデータブロック	1つの可変サイズの入りきらないデータブロック	複数の可変サイズの入りきらないデータブロック	オプションである固定サイズのインラインデータとオプションである最大64Kバイトのインラインデータ

図 12.17　Amoeba、V-System、Mach および Chorus の比較（次ページに続く）

ステムの設計目標、システムアーキテクチャ、重要で価値ある点は十分詳細にカバーされており、読者は分散オペレーティングシステムの設計原理に対する理解を深めることができる。4 つのシステムの比較についても述べた。

これらのシステムに関する説明から、オープンシステムアーキテクチャを持つマイクロカーネル準拠の分散オペレーティングシステムが将来優位になると結論することができる。商業的な成功のため、新しく開発されるどの分散オペレーティングシステムも UNIX とのバイナリ互換性を提供することが重要である。プロセス移動や負荷分散のような特徴はまだ研究プロジェクトに制限されている。これは主に、技術の現在の状態では、これらの特徴は実装し使用するには高価であり、約束された結果を与えることは証明されていないからである。したがって、この分野においては更に研究作業を行う必要がある。

特徴	Amoeba	V-System	Mach	Chorus
型付き/型なしメッセージ?	型なし	型なし	型付き/型なし	型なし
メッセージの宛先	プロセス	管理者 ID (プロセスまたはポート ID)	ポート	ポート
ポートまたはプロセスをロケーティングする機構	キャッシュのヒントとブロードキャスト	キャッシュのヒントとブロードキャスト	キャッシュのヒントとブロードキャスト	キャッシュのヒントとブロードキャスト
ローカルな IPC で書き込み時コピーを使う	いいえ	いいえ	はい	はい
グループ通信機能	高信頼で、順序保証のあるブロードキャスト機能	信頼性の度合いを選択できる柔軟性を有した順序保証のないブロードキャスト機能	なし	低信頼で、順序保証のないブロードキャスト機能
ネットワークワイドな IPC が何によってサポートされるか	カーネル	カーネル	ユーザレベルネットワークサーバ	ユーザレベルネットワークサーバ
サポートされているネットワークプロトコル	TCP/IP と FLIP	TCP/IP と VMTP	TCP/IP	TCP/IP
仮想メモリ機構がサポートされるか	いいえ	はい	はい	はい
同一ノード上のプロセスによって共有されるオブジェクトに対して書き込み時コピーがサポートされるか	いいえ	いいえ	はい	はい
外部のページ機構を使う	いいえ	いいえ	はい	はい
分散共有メモリに使われる一貫性のあるプロトコル	すべてに書き込み	問題固有な更新	読み取りのみ複製	読み取りのみ複製
ファイルモード	不変ファイル	可変ファイル	可変ファイル	可変ファイル
ファイル全体は連続して格納する必要がある?	はい	いいえ	いいえ	いいえ

図 **12.18** Amoeba、V-System、Mach および Chorus の比較（次ページに続く）

特徴	Amoeba	V-System	Mach	Chorus
遠隔ファイルアクセスモデル	データキャッシュモデル	データキャッシュモデル	データキャッシュモデル	データキャッシュモデル
データ転送の単位	ファイル	ブロック	ブロック	ブロック
ファイル共有のセマンティクス	不変共有ファイルセマンティクス	ユーザレベルのファイルサーバに依存	ユーザレベルのファイルサーバに依存	ユーザレベルのファイルサーバに依存
自動的なファイル複製	はい	いいえ	いいえ	いいえ
アクセス制御機構	一方向関数で暗号化されている資源のケーパビリティ	資源管理者により管理される ACL	ポートの権限（ポートケーパビリティ）	資源ケーパビリティと保護識別子
リアルタイムアプリケーションのサポート	いえ	はい	いいえ	はい
UNIX エミュレーション	ソースコード互換	バイナリ互換	バイナリ互換	バイナリ互換
オブジェクト指向言語とアプリケーションのサポート	いいえ	いいえ	いいえ	はい

図 **12.19**　Amoeba、V-System、Mach および Chorus の比較

12.8　演習問題

12.1 “オープン分散システム” とは何か？ 分散オペレーティングシステムの設計においてシステムのこの性格に影響を及ぼす最も重要な要因は何か？

12.2 Amoeba、V-System、Mach、Chorus の説明に基づいて、カーネル設計に対して現代の分散オペレーティングシステムが従っている傾向は何か？この方法を用いている主な理由は何か？このアプローチを使う利点はあるのか？

12.3 Amoeba の主なハードウェア構成要素とソフトウェア構成要素を挙げ、その機能を説明せよ。

12.4 Amoeba のケーパビリティ準拠のオブジェクト名前付け機構に対する次の質問に答えよ。

(a) オブジェクトのケーパビリティをどのようにして獲得するか？オブジェクトの所有者である場合と主体がオブジェクトの所有者でない場合の両方について考えよ。

(b) どのようにしてケーパビリティが権利のないユーザから守られるか？

(c) ケーパビリティはどのようにして偽造が難しくなっているか？

(d) オブジェクトにアクセスするとき、要求したモードでオブジェクトにアクセスできるかどうかの有効検査をどのようにして行うか？

(e) 他のものとオブジェクトを共有したいと思っているオブジェクトの所有者は、そのオブジェクトに対する制限されたアクセス権を相手に許すことがどのようにして可能なのか？

12.5 ケーパビリティ準拠のシステムでは、ケーパビリティが失われたオブジェクトに対しては決してアクセスすることができないであろう。Amoebaではこのような状況はどのようにして回避しているか?

12.6 Amoebaのプロセスモデルについて述べよ。

12.7 Amoebaにおけるグローバル変数とは何か？ なぜそれを使うのか？

12.8 Amoebaのプロセッサプールモデルにおいて、新しいプロセスを実行するのにどのようにしてプロセッサが選ばれるかについて説明せよ。

12.9 Amoebaでオブジェクトの複製に対して使われている方策について述べよ。

12.10 Amoebaは不変ファイルモデルを使っている。しかし、不変ファイルモデルを使用するファイルシステムでは、ファイルを修正するごとに、ファイルの新しい版が作成され、古い版が削除される。これによって、特に、頻繁に修正されるファイルに対しては、ファイルの修正が遅く複雑になる。Amoebaではこの問題をどのように解決しているか説明せよ。

12.11 Amoebaのグループ通信機能について述べよ。

12.12 次に示すものがAmebaにおいてどのように保証されるか説明せよ。

(a) 本物のクライアントだけがサーバのサービスを要求できる。

(b) 本物のサーバだけがクライアントからの要求を受信できる。

12.13 V-Systemの主なハードウェア構成要素とソフトウェア構成要素を挙げ、その機能について述べよ。

12.14 V-Systemはオブジェクト名前付けとオブジェクト管理の機構を統合している。このアプローチは分散オペレーティングシステムの設計においてどのような利点を持つか？

12.15 V-Systemにおいてプロセス管理作業でカーネルのジョブを最小化するための機構について述べよ。

12.16 V-SystemのUIOインタフェースについて述べよ。この機能はデバイス管理作業をどのように簡素化するかについて説明せよ。

12.17 V-Systemのマルチキャスト機能について述べよ。V-Systemの設計においてこの機能の実際的なアプリケーションの幾つかについて説明せよ。

12.18 V-Systemでは、要求を送信し応答を受信する送信側の作業は1つのSendプリミティブに結合されている。このアプローチはどのような利点を提供

するか？Send_request と Get_reply という、1つは要求を送信するための、もう1つは応答を受信するための2つの呼び出しを持つことは、より優れ、かつ単純でないのか？

12.19 Mach の書き込み時コピーの機構がシステム内で多くのデータの移動を削減する能力をどのように提供するかについて説明せよ。

12.20 Mach において、多くのカーネルの作業は、透過的な共有ライブラリを置いていることで、カーネルからクライアントのアドレス空間に直接的に移動される。この設計が提供する利点とは何か？

12.21 スレッド (又はプロセス) の状態 (一時停止／実行中) を追跡するために、Mach は各スレッド (又はプロセス) に関連した一時停止カウンタを使っている。2つの状態 (一時停止と実行中) しかないので、1つのビットでこの目的を満たすのに十分である。この目的のためにビットではなくカウンタを使っている理由について説明せよ。

12.22 Mach ではどのようにしてプロセスに属するスレッドが同期されるか述べよ。

12.23 Mach のスレッドスケジューリングの方式に対する次の質問に答えよ。

(a) “ヒント” 変数の利用とは何か？

(b) “カウント” 変数の利用とは何か？

(c) 各 CPU に対して1つの実行キューの代わりに2つの実行キューを持つ必要性とは何か？

(d) 2つのキューのどちらにも実行を待っているスレッドがない場合、何が起きるか？

(e) 負荷が軽いときはシステムは高い効率性を提供し、負荷が高いときは短い要求に対する良い応答を提供することはどのようにして保証されているか？

(f) 高い優先度スレッドによる CPU の独占をどのようにして避けているか？

(g) システムの負荷が突然大きくなった場合、CPU の公平な利用をどのようにして保証しているか？

(h) 実行準備ができているスレッドを検索する前に、グローバル実行キューをロックするが、ローカル実行キューはロックしないのはなぜか？

12. 24 各プロセスの仮想アドレス空間を管理するために共通に用いられているアプローチは、カーネル中に0から最も大きなページ番号のページまでのリニアページテーブルを保持することである。なぜ Mach ではこのアプローチが使われていないのか説明せよ。Mach においてプロセスの仮想アドレス空間を管理するために用いられているアプローチについて述べよ。

12.25 Mach の外部メモリ管理者とは何か？それはどのように実現されているか？メモリオブジェクトを管理し、メモリオブジェクトのデータに対するユーザアクセスの要求を満たすために、メモリ管理者とカーネルはお互いにどのようにかかわるかについて述べよ。

12.26 Mach の概念である外部メモリ管理者は分散共有メモリ機能を実装するためにどのように使われているか述べよ。

12.27 メモリ共有のために Mach で使われている書き込み時コピーと制御継承機構について述べよ。それぞれの機構の実際の利用を示す例を与えよ。

12.28 Mach のポート準拠の IPC 機構に対する次の質問に答えよ。

(a) なぜポートはスレッドごとに基づいてではなく、プロセスごとに基づいて追跡されるのか？

(b) アプリケーションは、送信側がポートに対して n 回だけメッセージを送ることが可能であることを要求する。これはどのように実現できるか？

(c) receive 権利のポートケーパビリティを持つプロセスがそのケーパビリティを別のプロセスにメッセージに入れて送る場合、何が起きるか？

(d) どのようにしてポートは保護されるか？

(e) receive 権利を持つポートケーパビリティを所有しているプロセスが終了するか殺された場合、何が起きるか？

(f) もし、送信側が既に殺された受信プロセスのポートにメッセージを送った場合、何が起きるか？

(g) キューが一杯であるポートにメッセージが到着した場合、何が起きるか？

(h) ポート集合とは何か？この機能の実際の利用を挙げよ。

(i) ポート集合にメッセージを送ることはどうして可能なのか？

(j) ポート集合に属しているポートをメッセージを直接受信するに使用することはどうしてできないのか？

12.29 Mach のメッセージパッシング機構においてメッセージ内データとメッセージ外データの違いを述べよ。メッセージ外データは送信側から受信側へどのように転送されるか説明せよ。

12.30 Mach において、異なったノードに位置する 2 つのプロセス間で、どのようにしてメッセージ交換が透過的に行われるかを説明せよ。

12.31 Mach における MIG の役割は何か。これがどのようにアプリケーションプログラマの仕事を単純にするかについて説明せよ。

12.32 Mach の UNIX エミュレーションでは、ファイル入出力に関するシステム呼び出しは、本章で述べた基本的な方法とは異なった方法で実装されてい

る。どのようにファイル入出力システム呼び出しが実装されているかを調査し、異なった実装方式が用いられてい理由を調べよ。

12.33 Chorus では、どのようにしてリアルタイムアプリケーションをサポートしているかについて説明せよ。

12.34 Chorus では、サーバ (サブシステム) が共有カーネルアドレス空間にも、私有ユーザアドレス空間にも存在できる。Chorus のこの特徴の相対的な長所、短所を説明せよ。

12.35 Chorus は、システムの特定のノードのハードウェア構成要素に合わせて、システムソフトウェアを動的に構成する柔軟性を、どのようにして提供しているか。

12.36 Chorus で提供されるセキュリティ機構は何か。これらの機構はどのタイプのセキュリティを提供するか。

12.37 Chorus において、3 つのプロセスタイプが用いられる理由を説明せよ。

12.38 Chorus におけるスレッドスケジューリング方式を説明せよ。その長所、短所は何か。

12.39 Chorus における通信の、同報モード、機能モード及び選択的機能モードの違いを説明せよ。各通信モードの実用的な用途を挙げよ。

12.40 Chorus のポート移動機能を説明せよ。これは何の役に立つか。

12.41 Chorus のポートグループとは何か。この機能の実用的な用途を 2 つ挙げよ。

12.42 Chorus において、UNIX プロセスが遠隔ノード上でどのように生成されるかを説明せよ。

12.43 UNIX エミュレーションに関して、Mach で採られた方法と Chorus で採られた方法の相対的な長所、短所について説明せよ。

12.44 Chorus では、オブジェクト指向機能がどのようにサポートされているかを説明せよ。この機能は、同じマシン上の UNIX エミュレーション機能と一緒に提供することが可能か。その理由も示せ。

参考文献

[Abrossimov et al. 1992] Abrossimov, A., Armand, F., and Ortega, M., "A Distributed Consistency Server for the CHORUS System," In: Proceedings of the USENIX SEDMS 3 Symposium on Experience with Distributed and Multiprocessor Systems, USENIX Association, Berkeley, CA, pp. 129-148 (1992).

[Accetta et al. 1986] Accetta, M., Baron, R., Golub, D., Rashid, R., Tevanian, A., and Young, M., "Mach: A New Kernel Foundation for UNIX Development," In: Proceedings of the Summer 1986 USENIX Technical Conference, USENIX Association, Berkeley, CA, pp. 93-112 (July 1986).

[Almes et al. 1985] Almes, G. T., Black, A. P., Lazowska, E. D., and Noe, J. D., "The Eden System: A Technical Review," IEEE Transactions on Software Engineering, Vol. SE-11, No. 1, pp. 43-59 (January 1985).

[Andrews et al. 1987] Andrews, G. R., Schlichting, R. D., Hayes, R., and Purdin, T. D. M., "The Design of the Saguaro Distributed operating System," IEEE Transactions on Software Engineering, Vol. SE-13, No.1, pp.104-118 (1987).

[Armand et al. 1986] Armand, F., Gien,m M., Guillemont, M., and Leonard, p., "Towards a Distributed UNIX System—The CHORUS Approach," In: Proceedings of the Autumn 1986 EUUG Conference, USENIX Association, Berkeley, CA, pp. 413-431 (September 1986).

[Armand et al. 1989] Armand, F., Gien, M., Herrman, F., and Rozier, M., "Distributing UNIX Brings It Back to Its Original Virtues," In: Proceedings of the Workshop on Experiences with Building Distributed and Multiprocessor Systems, pp. 153-174 (October 1989).

[Babaoglu 1990] Babaoglu, O., "Fault-Tolerant Computing Based on Mach," Operating Systems Review, Vol. 24, No. 1, pp. 27-39 (1990).

[Bal et al. 1992] Bal, H. E., Kaashoek, F., and Tanenbaum, A. S., "Orca: A Language for Parallel Programming of Distributed Systems," IEEE Transactions oon Softwre Engineering, Vol. SE-18, No. 3, pp.190-205 (1992).

[Banino and Fabre 1982] Banino, J. S., and Fabre, J. C., "Distributed Coupled Actors: A CHORUS Proposal for Reliability," In: Proceedings of the 3rd International Conference on Distributed Computing Systems, IEEE Press, New York, NY, p. 7 (October 1982).

[Banino et al. 1985] Banino,k J. S., Fabre, J. C., Guillemont, M., Morisset, G., and Zimmenrmann, H., "Some Fault Tolerant Aspects of the CHORUS Distribute System," In: Proceedings of the 5th International Conference on Distributed Computing systems, IEEE Press, New York, NY (May 1985).

[Baron et al. 1985] Baron, R., Rashid, R., Siegel, E., Tevanian, A., and Young, M., "Machi-1: An Operating Environment for Large-Scale Multiprocessor Applications," IEEE Software, Vol.2, pp. 65-67 (1985).

[Batllivala et al. 1992] Batlivala, N., Gleeson, B., Hamrick, J., Lurndal, S., Price D., Soddy, J., and Abrossimov, V., "Experience with SVR4 over CHORUS," In: Proceedings of the USENIX Workshop on Microkernels and Other Kernel Architectures, USENIX Association, Berkeley, CA, pp. 223-341 (1992).

[Berglund 1986] Berglund, E. J., "An Introduction to the V-System," IEEE MICRO, pp. 35-52 (August 1986).

[Bever et al. 1993] Bever, M., Geihs, K., Heuser, L., Muhlhauser, M., and Schill, A., "Distributed Systems, OSF DCE, and Beyond," In: A. Schill (Ed.), DCE—The OSF Distributed Computing Environment, Springer-Verlag, Berlin, pp. 1-20 (1993).

[Black 1990] Black, D. L., "Scheduling Support for Concurrency and Parallelism in the Mach Operating System, IEEE Computer, Vol. 23, No. 5,pp. 35-43 (1990).

[Black et al. 1992] Black, D. L., Golub, D. B., Lulin, D. P., Rashid, R. F., Draves, R. P., Dean, R. W., Forin, A., Barrera, J., Tokuda, H., Malan, G., and Bohman, D., "Microkernel Operating System Architecture and Mach," In: Proceedings of the USENIX Workshop on Microkernels and Other Kernel Architectures, USENIX Association, Berkeley, CA, pp. 11-30 (1992).

[Boykin et al. 1993] Boykin, J., Kirschen, D., Langerman, A., and LOVerso, S., Programming Under Mach, Addison-Wesley, Reading, MA (1993).

[Boykin and Langerman 1990] Boykin, J., and Langerman, A., "Mach/4.3BSD: A Conservative Approach to Parallelization," Computing Systems Journal, Vol. 3, pp. 69-99 (1990).

[Champine et al. 1990] Champine, G. A., Geer, Jr., D. E., and Ruh, W. N., "Project Athena as a Distributed Computer system," IEEE Computer, pp. 40-50 (September 1990).

[Cheriton 1984] Cheriton, D. R., "The V Kernel: A Software Base for Distributed Systems," IEEE Software, Vol. 1, No. 2, pp. 19-42 (April 1984).

[Cheriton 1987] Cheriton, D. R., "UIO: A Uniform I/O Interface for Distributed Systems," ACM Transactions on Computer Systems, Vol. 5, No. 1, pp. 12-46 (1987).

[Cheriton 1988] Cheriton, D. R., "The V Distributed System," Communications of the ACM, Vol. 31, No. 3, pp. 314-333 (1988). ©ACM, Inc., 1988.

[Cheriton and Mann 1989] Cheriton, D. R., and Mann, T. P., "Decentralizing a Global Naming Service for Improved Performance and Fault Tolerance," ACM Transactions on Computer Systems, Vol. 7, No. 2, pp.147-183 (1989).

[Cheriton et al. 1979] Cheriton, D. R., Malcolm, M. A., Melen, L. S., and Sager, G. R., "Thoth, a Portable Real-Time Operating System," Communications of the ACM, Vol. 22, No. 2, pp. 105-115 (February 1979).

[Cheriton et al. 1990] Cheriton, D., Whitehead, G., and Sznyter, E., "Binary Emulation of UNIX Using the V Kernel," In: Proceeding of the Summer USENIX Conference, USENIX Association, Berkeley, CA, pp.73-85 (1990).

[Coulouris et al. 1994] Coulouris, G. F., Dollimore, J., and Kindberg, T., Distributed Systems Concepts and Design, 2nd ed., Addison-Wesley, Reading, MA (1994).

[Dasgupta et al. 1991] Dasgupta, P., LeBlanc, R. J., Ahamad, M., and Ramachandran, U., "The Clouds Distributed Operating system," IEEE Computer, Vol. 24, No. 11, pp. 34-44 (1991).

[Dean and Armand 1992] Dean, R., and Armand, F., "Data Movement in Kernelized Systems," In: Proceedings of the USENIX Workshop on Microkernels, USENIX Association, Berkeley, CA (1992).

[Douglis et al. 1991] Douglis, F., Ousterhout, J. K., Kaashoek, M. F., and Tanenbaum, A. S., "A Comparison of Two Distributd Systems: Amoeba and Sprite," Computing Systems Journal, Vol. 4, pp. 353-384 (1991).

[Draves 1990] Draves, R. P., "A Revsed IPC Interface," In: Proceedings of the USENIX Mach Workshop, USENIX Association, Berkeley, CA, pp.101-121 (October 1990).

[Draves et al. 1989] Draves, R. P., Jones, M. B., and Thompson, M. R., "MIG—The Mach Interface Generator," Technical Report, Department of Computer Science, Carnegie Mellon University (1989).

[Duff 1990] Duff, T., "Rc—A Shell for Plan 9 and UNIX Systems," In: Proceedings of the Summer 1990 UKUUG Conference, USENIX Association, Berkeley, CA, pp. 21-33 (July 1990).

[Finkel et al. 1989] Finkel, R., Scott, M. L., Kalsow, W. K., Artsy, Y., and Chang, H. Y., "Experience with Charlotte: Simplicity and Function in a Distributed Operating System," IEEE transactions on Software Engineering, Vol. SE-15, No. 6, pp. 676-685 (1989).

[Fitzgerald and Rashid 1986] Fitzgerald, R., and Rashid, R. F., "The Integration of Virtual Memory Management and Interprocess Communication in Accent," ACM Transactions on Computer Systems, Vol. 4, No.2, pp.147-177 (1986).

[Gien and Grob 1992] Gien, M., and Grob, L., "Microkernel Based Operating Systems: Moving UNIX on to Modern System Architectures," In: Proceedings of the UniForum'92 Conference, USENIX Association, Berkeley, CA, pp. 43-55 (1992).

[Golub et al. 1990] Golub, D., Dean, R, Forin, A., and Rashid, R., "UNIX as an Application Program," In: Proceedings of the Summer 1990 USENIX Conference, USENIX Association, Berkeley, CA, pp. 87-95 (June 1990).

[Guillemont 1982] Guillemont, M., "The CHORUS Distributed Computing System: Design and Implementation," In: Proceedings of the International Symposiium on Local Computer Networks, pp. 207-223 (April 1982).

[Jones and Rashid 1986] Jones, M. B., and rashid, R. F., "Mach and Matchmaker: Kernel and Language Support for Object-Oriented Distributed Systems," In: Proceedings of OOPSLA'86, Association for Computing Machinery, New York, Ny, pp. 67-77 (September 1986).

[Kaashoek and tanaenbaum 1991] Kaashoek, M. F., and Tanenbaum, A. S., "Group Communication in the Amoeba Distributed Operating system," In: Proceedings of the 11th International Conference on Distributed Computing systems, IEEE Press, New York, NY, pp. 222-230 (May 1991).

[Keefee et al. 1985] Keefee, D., Tomlinson, G. M., Wand, I. C., and Wellings, A. J., PULSE: An Ada-Based Distributed Operating System, Academic Press, San Diego, CA (1985).

[Lea et al. 1991] Lea, R., Amaral, P., and Jacquemot, C., "COOL-2: An Object-Oriented Support Platform Built Above the Chorus Microkernel," In: Proceedings of the International Workshop on Object-Oriented Systems, pp. 51-55 (1991).

[Lea et al. 1993] Lea, R., Jacquemot, C., and Pillevesse, E., "COOL: System Support for Distributed Programming," Communications of the ACM, Vol. 36, No. 9, pp. 37-46 (September 1993).

[Levine 1987] Levine, P. H., "The DOMAIN System," In: Proceedings of the 2nd ACM SIGOPS Workshop on Making Distributed Systems Work, Operating Systems Review, Vol. 21, No. 1, pp. 49-84 (1987).

[Miller et al. 1987] Miller, B. P., Presotto, D. L., and Powell, M. L., "DEMOS/MP: The Development of a Distributed Operating System," Software—Practice and Experience, Vol. 17, No. 4, pp. 277-290 (1987).

[Milojcic 1994] Milojcic, D. S., Load Distribution, Implementation for the Mach Microkernel Verlag Vieweg, Wiesbaden (1994).

[Morris et al. 1986] Morris, J. H., Satyanarayanan, M., Conner, M. H., Howard, J. H., Rosenthal, D. S. H., and Smith, F. D., "Andrew: A Distributed Personal Computing Environment," Communications of the ACM, Vol. 29, No. 3, pp. 184-201 (1986).

[Mullender and Tanenbaum 1986] Mullender, S. J., and Tanenbaum, A. S., "The Design of a Capability-Based Distributed Operating System," The Computer Journal, Vol. 29, No. 4, pp. 289-299 (1986).

[Mullender et al. 1990] Mullender, S. J., Van Rossum, G., Tanenbaum, A. S., Vanesse, R., and Van Staverene, H., "Amoeba: A Distributed Operating System for the 1990s," IEEE Computer, Vol. 23, No. 5, pp.44-53 (1990).

[Needham and Herbert 1982] Needham, R. M., and Herbert, A. J., The Cambridge Distributed Computing System, Addison-Wesley, Reading, MA (1982).

[Nelson and Leach 1984] Nelson, D. L., and Leach, P. J., "The Architecture and applications of the Apollo DOMAIN," IEEE Computer Graphics and Applications (April 1984).

[Orman et al. 1993] Orman, H., Menze, E., O'Malley, S., and Peterson, L., "A Fast and General Implementation of Mach IPC in a Network," In: Proceedings of the 3rd USENIX MACH Workshop, USENIX Association, Berkeley, CA (April 1993).

[Ousterhout et al. 1988] Ousterhout, J. K., Cherenson, A. R., Douglis, F., Nelson, M. N., and Welch, B. B., "The Sprite Network Operating System," IEEE Computer, Vol. 21, No. 2, pp. 23-36 (1988).

[Pike et al. 1990] Pike, R., Presotto, D., Thompson, K., and Trickey, H., "Plan 9 from Bell Labs," In: Proceedings of the Summer 1990 UKUUG (UK Unix Users Group) Conference, USENIX Association, Berkeley, CA, pp. 1-9 (July 1990).

[Popek and Walker 1985] Popek, G., and Walker, B., The LOCUS Distributed System Architecture, MIT Press, Cambridge, MA (1985).

[Pountain 1994] Pountain, D., "The Chorus Microkernel," BYTE, pp.131-136 (January 1994).

[Presotto et al. 1991] Presotto, D., Pike, R., Thompson, K., and Trickey, H., "Plan 9, A Distributed System," In: Proceedings of the Spring 1991 EurOpen Conference, EurOpen, Hertfordshire, UK, pp. 343-50 (May 1991).

[Rashid 1986] Rashid, R. F., "From RIG to Accent to Mach: The Evolution of a Network Operating System," In: Proceedings of the Fall Joint Computer Conference, AFIPS, pp. 1128-1137 (November 1986).

[Rashid 1987] Rashid, R. F., "Mach: A New Foundation for Multiprocessor Systems Development," In: Proceedings of COMPCON'87—Digest of Papers, IEEE Press, New York, NY, pp. 192-193 (1987).

[Rashid and Robertson 1981] Rashid, R. F., and Robertson, G., "Accent: A Communication Oriented Network Operating System Kernel," In: Proceedings of the 8th ACM Symposium on Operating Systems Principles, Association for Computing Machinery, New York, Ny, pp. 64-75 (December 1981).

[Rashid et al. 1988] Rashid, R., Tevanian, A., young, M., Golub, D., Baron, R., Black, D., Bolosky, W. J., and Chew, J., "Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architecture," IEEE Transactions on Computers, Vol. C-37, No. 8, pp. 869-908 (1988).

[Rozier and Legatheaux 1986] Rozier, M., and Legatheaux, J. M., "The Chorus Distributed Operating System: Some Design Isssues," In: Y. Parker et al. (Eds.), Distributed Operating Systems: Theory and Practice, NATO ASI Series, Vol. F28, Springer-Verlag, New York, NY, pp.261-289 (1986).

[Rozier et al. 1988] Rozier, M., Abrossimov, V., Armand, F., Boule, I., Gien, M., Guillemont, M., Herrmann, F., Kaiser, C., Leonard, P., Langlois, S., and Neuhauser, W., "Chorus Distributed Operating System," Computing Systems Journal, Vol. 1, No. 4, pp. 305-379 (1988).

[Satyanarayanan et al. 1990] Satyanarayanan, M., Kistler, J. J., Kumar, P., Okasaki, M. E., Siegel, E. JH., and Steere, D. C., "Coda: A Highly Available File System for a Distributed Workstation Environment," IEEE Transactions on Computers, Vol. 39, No. 4, pp. 447-459 (1990).

[Schantz et al. 1986] Schantz, R. E., Thomas, R. H., and Bono, G., "The Architecture of the Cronus Distributed Operating System," In: Proceedings of the 6th International Conference on Distributed Computing Systems, IEEE Press, New York, NY, pp. 250-259 (1986).

[Schroeder et al. 1984] Schroeder, M. D., Birrell, A. D., and Needham, R., M., "Experience with Grapevine: The Growth of a Distributed System," ACM Transactions on Computer Systems, Vol. 2, No. 1, pp. 3-23 (1984).

[Shrivastava et al. 1991] Shrivastava, S., Dixon,k G. N., and Parrington, G. D., "An Overview of the Arjuna Distributed Programming System," IEEE Software, pp. 66-73 (January 1991).

[Silberschatz and Galvin 1994] Silberschatz, A., and Galvin, P. B¿, Operating Systems Concepts, 4th ed., Addison-Wesley, Reading, MA (1994).

[Sinha et al. 1991] Sinha, P. K., Maekawa, M., Shimizu, K., Jia, X., Ashihara, H., Utsunomiya, N., Park, K. S., and Nakano, H., "The Galaxy Distributed Operating System," IEEE Computer, Vol. 24, No. 8, pp. 34-41 (1991).

[Sinha et al. 1994] Sinha, P. K., Maekawa, M., Shimizu, K., Jia, X., Ashihara, H., Utsunomiya, N., Park, K. S., and Nakano, H., "The Architectural Overview of the Galaxy Distributed Operating System," In: T. L. Casavant and M. Singhal (Eds.), Readings in Distributed Computing Systems, IEEE Computer Society Press, Los Alamitos, CA, pp. 327-345 (1994).

[Swinehart et al. 1986] Swinehart, D., et al., "A Structural View of the Cedar Programming Environment," ACM Transactions on Programming Languages and Systems, Vol. 8, No. 4, pp. 419-490 (1986).

[Tanenbaum 1995] Tanenbaum, A. S., Distributed Operating Systems, Prentice-Hall, Engliwood Cliffs, NJ (1995).

[Tanengbaum and Van Renesse 1985] Tanenbaum, A. S., and Van Renesse, R., "Distributed Operating Systems," ACM Computing Surveys, Vol. 17, No. 4, pp. 419-470 (1985).

[Tanenbaum et al. 1990] Tanenbaum, A. S., Van Renesse, R., Staveren, H. Van, Sharp, G. J., Mullender, S. J., Jansen, J., and Van Rossum, G., "Experiences with the Amoeba Distributed Operating System," Communications of the ACM, Vol. 33, pp. 46-63 (1990).

[Theimer et al. 1985] Theimer, M. M., Lantz, K. A., and Cheriton, D. R., "Preemptablee Remote Execution Facilities for the V-System," In: Proceedings of the 10th ACM Symposium on Operating Systems Principles, Association for Computing Machinery, New York, NY, pp. 2-12 (December 1985).

[Tokuda et al. 1990] Tokuda, H., Nakajima, T., and Rao, P., "Real-Time Mach: Towards a Predictable Real-Time System," In: Proceedings of the USENIX Mach Workshop, USENIX Association, Berkeley, CA, pp.73-82 (October 1990).

[Tripathi 1989] Tripathi, A. R., "An Overview of the Nexus Distributed Operating SystemDesign," IEEE Transactions on Software Engineerign, Vol. SE-15, No. 6, pp. 686-695 (1989).

[Wilkes and Needham 1980] Wilkes, M. V., and Needham, R. M., "The Cambridge Model Distributed System," Operating Systems Review, Vol.14, No.1, pp.21-29 (1980).

[Zimmermann et al. 1981] Zimmermann, H., Banino, j. S., Caristan, A., Guillemont, M., and Morisset, G., "Basic Concepts for The Support of Distributed Systems: The CHORUS Approach," In: Proceedings of the 2nd International Conference on Distributed Computing Systems, IEEE Press, New York, NY, pp. 60-66 (1981).

インターネット参考文献

Amoeba に関する参考文献は、以下のところで見つけることができる。

http://www.cs.vu.nl/vakgroepen/cs/amoeba_papers.html
ftp:ftp.cs.umanitoba.ca/pub/bibliographies/Distributed/amoeba.html

V-Systemのみを扱った参考文献は見つけられなかったが、以下の参考文献にはこのシステムに関する記述が見られる。
http://www-dsg.stanford.edu/Publications.html
ftp:ftp.cs.umanitoba.ca/pub/bibliographies/Os/IMMD_IV.html
ftp:ftp.cs.umanitoba.ca/pub/bibliographies/Os/os.html

Machに関する参考文献は、以下のところで見つけることができる。
ftp:ftp.cs.umanitoba.ca/pub/bibliographies/Distributed/Mach.html
http://www.cs.cmu.edu/afs/cs/project/mach/public/www/doc/publications.html

Machの刊行リストは、以下のところで見つけることができる。
http://www.cs.cmu.edu/afs/cs/project/mach/public/www/doc/books.html

Real Time Machに関する参考文献は、以下のところで見つけることができる。
http://www.cs.cmu.edu/afs/cs/project/mach/public/www/doc/rtmach.html
http://www.cs.cmu.edu/afs/cs/project/art-6/www/publications.html

Mach-US(CMUのMachプロジェクトの一環として開発されたオペレーティングシステム)に関する参考文献は、以下のところで見つけることができる。
http://www.cs.cmu.edu/afs/cs.cmu.edu/project/mach/public/www/projects/mach_us.html

Chorusに関する参考文献は、以下のところで見つけることができる。
ftp:ftp.cs.umanitoba.ca/pub/bibliographies/Distributed/chorus.html

索　引

<翻訳者略歴>

翻訳主幹

水野　忠則（みずの ただのり）

1968年　名古屋工業大学経営工学科卒業
現　在　静岡大学情報学部教授・工学博士

井手口哲夫（いでぐち てつお）

1972年　電気通信大学通信工学科卒業
現　在　愛知県立大学情報科学部教授・工学博士

福岡　久雄（ふくおか ひさお）

1978年　神戸大学大学院工学研究科修了
現　在　三菱電機株式会社 通信機製作所・工学博士

宮西洋太郎（みやにし ようたろう）

1968年　神戸大学大学院工学研究科修了
1997年　静岡大学大学院電子科学研究科修了
現　在　三菱電機株式会社 情報システム製作所・工学博士

佐藤　文明（さとう ふみあき）

1986年　東北大学大学院工学研究科博士前期課程修了
現　在　静岡大学情報学部助教授・工学博士

分散オペレーティングシステム　ISBN4-87653-008-4

2000年1月15日　初版印刷
2000年4月25日　初版発行

著　者　Pradeep K. Sinha
訳　者　水野　忠則（翻訳主幹）
井手口哲夫・福岡　久雄
宮西洋太郎・佐藤　文明
発行者　村越　勝弘
発行所　株式会社 科学技術出版
〒113-0033
東京都文京区本郷2丁目16-12
電　話　03-3813-6920　FAX：03-3818-7888
E-mail　info@kagaku.co.jp
URL　http://www.kagaku.co.jp/
印　刷　三美印刷株式会社
製　本　松岳社

Printed in Japan

2205